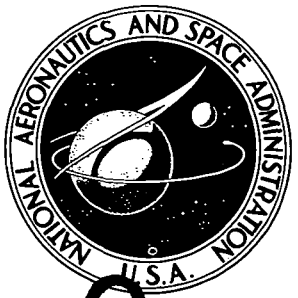
NASA TECHNICAL NOTE



NASA TN D-7200

NASA TN D-7200

PROPOSAL FOR HIERARCHICAL DESCRIPTION OF SOFTWARE SYSTEMS

by H. Trauboth

George C. Marshall Space Flight Center
Marshall Space Flight Center, Ala. 35812

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • MARCH 1973

TECHNICAL REPORT STANDARD TITLE PAGE

1. REPORT NO.	2. GOVERNMENT ACCESSION NO.	3. RECIPIENT'S CATALOG NO.
NASA TN D-7200		

4. TITLE AND SUBTITLE	5. REPORT DATE
Proposal for Hierarchical Description of Software Systems	March 1973
	6. PERFORMING ORGANIZATION CODE
7. AUTHOR(S) H. Trauboth	8. PERFORMING ORGANIZATION REPORT #
9. PERFORMING ORGANIZATION NAME AND ADDRESS George C. Marshall Space Flight Center Marshall Space Flight Center, Alabama 35812	10. WORK UNIT NO.
	11. CONTRACT OR GRANT NO.
	13. TYPE OF REPORT & PERIOD COVERED
12. SPONSORING AGENCY NAME AND ADDRESS National Aeronautics and Space Administration Washington, D. C. 20546	Technical Note
	14. SPONSORING AGENCY CODE

15. SUPPLEMENTARY NOTES

Prepared by Computation Laboratory,
Science and Engineering

16. ABSTRACT

The programming of digital computers has developed into a new dimension full of difficulties, because the hardware of computers has become so powerful that more complex applications are entrusted to computers. The costs of software development, verification, and maintenance are outpacing those of the hardware and the trend is toward further increase of sophistication of application of computers and consequently of sophistication of software. To obtain better visibility into software systems and to improve the structure of software systems for better tests, verification, and maintenance, a clear, but rigorous description and documentation of software is needed. The purpose of this report is to extend the present methods in order to obtain a documentation that better reflects the interplay between the various components and functions of a software system at different levels of detail without losing the precision in expression. This is done by the use of block diagrams, sequence diagrams, and cross-reference charts. In the appendices, examples from an actual large software system, i.e. the Marshall System for Aerospace Systems Simulation (MARSYAS), are presented. The proposed documentation structure is apt to automation of updating significant portions of the documentation for better software change control. This report should also stimulate research into new practical methods and principles for the development, verification, and maintenance of complex software systems.

Note: Date of the manuscript is October 1, 1971.

17. KEY WORDS	18. DISTRIBUTION STATEMENT

19. SECURITY CLASSIF. (of this report)	20. SECURITY CLASSIF. (of this page)	21. NO. OF PAGES	22. PRICE
Unclassified	Unclassified	138	$3.00

MSFC - Form 3292 (May 1969)

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Concluded)

LIST OF TABLES

PROPOSAL FOR HIERARCHICAL DESCRIPTION OF SOFTWARE SYSTEMS

INTRODUCTION

The programming of digital computers has developed into a new dimension full of difficulties, because the hardware of computers has become so powerful that more complex applications are entrusted to the computer. While 15 years ago, a program was considered large if it contained 1000 instructions, today, program systems with 100 000 instructions are no scarcity anymore. The costs of software development are outpacing those of hardware and the trend is toward further increase of sophistication of application of computers and consequently of sophistication of software [1]. In many large-scale, real-time systems, the computer and its programs have become a vital part of the total system. For instance, the flight computer of the Space Shuttle and the tracking computer of an antiballistic missile are now responsible for the essential control functions during the launch and flight of these space vehicles. Instead of simple programs, we have now to deal with complex software systems. The complexity and size have caused severe problems in the development, verification, and maintenance of software systems.

The reliability of software is often so low that schedules of projects which depend on computer software become unpredictable and the user is losing his confidence in computer technology. In real-time systems where high reliability is necessary, such as in an avionics system for a space vehicle or a structural test facility, software checkout and verification have become very expensive and time consuming. The verification costs for such critical real-time applications exceed often 50 percent of the software development costs [2].

The visibility into a large software system and consequently the confidence in an estimate of the manpower and time necessary for development are poor. Schedule slippages in software development of about 100 to 200 percent of the planned time are not unusual. Management of software systems often lacks the information and understanding it needs to make intelligent decisions at various levels and phases.

Many of these problems stem from the fact that computer technology and particular software is a very new technology; it is still more of an art than

a science. The software design does not use many generally proven principles. Each computer program is written as a new entity, seldom using other basic programs as building blocks. Not many standards exist (except flow chart symbols and Backus-Naur notation) for describing software. If 10 different programmers wrote the same computer program, they would describe it in 10 different ways. One might compare the software state-of-the-art with the state of electronics around World War I, when no circuit analysis and synthesis techniques were known. Then, electronic circuits were designed by pure experimentation with the assistance of Ohm's and Kirchoff's laws. However, now, techniques to describe, analyze, and synthesize the various functions of complex electronic circuits are so powerful that they allow to automate many design steps. In contrast, software still lacks a methodology based on standard symbolics and mathematically well-defined relationships which would give insight into complex software systems.

Although presently available documentation describes computer programs in detail, it fails to give insight into the system at a higher level and to show explicitly the interrelationships between the various software components. Thereby, it hampers the manager to plan properly and make intelligent decisions during the development of software.

The purpose of this report is to improve the present situation by providing guidelines for a more rigorous description and documentation of software based on commonly practiced ways of describing software [11]. The present methods are simply extended to obtain a more-engineer-like and systematic description of software functions. This extension was stimulated by reading the documentation of a large software system which is the Marshall System for Aerospace Systems Simulation (MARSYAS) that was developed by Computation Laboratory of Marshall Space Flight Center. An improved software description should be structured to better serve the systems engineering of software. Hence, the documentation should reflect the interplay between the various components and functions of the software system at different levels of detail without loosing the precision in expression. The information that will be used to describe the software is already available in present documentations; however, it will be presented in different forms using block diagrams and charts similar to those familiar to engineers. These different forms should assist the systems designer to gain insight into the interrelationships of the various functional units of a software system. Thereby, he should be enabled to better perform checkout, verification, overlays of data tables, and modifications. The structure of the documentation should also allow easy automatic generation and up-dating of portions of the documentation by a computer.

The human mind is rather limited in comprehending and memorizing large intermeshed systems. He can follow small steps in a logical sequence which leads to an understanding of a larger entity. Therefore, human organizations such as the Government are built in hierarchies using small components and groups of these components which can be overlooked. Even the human language accounts for this hierarchical thinking in its vocabulary. Also, complex software systems should be structured in hierarchies of modules. Then, the software can be described by its individual components and their interrelationships. The functions of these small components can be understood and their interplay be overviewed. The smallest components which cannot be divided anymore are the foundation of the hierarchy. Their functions have to be described precisely so that all higher-level components can be based on them. It is like building a house on a firm foundation in order not to collapse. The documentation of software systems should reflect this hierarchical structure of the software.

This report does not unveil any new break-through method for the description of software systems. To obtain fundamentally new methods and principles which are useful for software development, much more intensive basic research is required. Hopefully, this report might stimulate more original thinking and might assist in activating funds to pursue the extremely needed research.

This report has two sections: Section I shows how tables, subprograms, and their interrelationships should be described, and Section II outlines the various documents of the documentation and includes some thoughts about verification of software and up-dating and automating of documentation. The appendices present examples from MARSYAS to aid the explanation.

A software system is viewed as a highly coupled logical network of basic components. We distinguish between two types of components: (1) the subprograms which contain the instructions, and (2) the tables, including indicators and buffers, which contain the data to be processed by the subprograms.

SECTION I. GUIDELINES FOR SOFTWARE DESCRIPTION

A. Tables

1. General. The information to be processed by the computer is stored in groups of memory cells. These memory cells may be physically located in the main memory (i.e., the core) or in other memory devices such as index registers, magnetic drum, disc, or tape. In this report, we do not concern ourselves with the physical location, rather with the logical location of tables, the identification of data, and the description of the contents of logical memory cells.

A table is defined here as a collection of data in which each data item is identified by a label or name, by its position relative to other data items, or by some other means [3]. Data items mean the individual member of a set of data. In the following, we will use the term table also for lists, arrays, dictionaries, directories, etc. A collection of tables will be called a file. The tables are, in general, of variable length and are structured in a hierarchical way to better access and retrieve the contents of the tables. A simple, though consistent, nomenclature should be used to identify unambiguously the data items to be accessed. Names that are used to address or label a data item should be chosen in such a way that they give a good and concise indication of the meaning of the data item. It is important for the reader of the documentation on tables to obtain a visually good picture of the structure and contents of the tables so that he can memorize them easily. Also, the shorthand notation should be as close as possible to the English language.

The description of the tables should also produce a clear and easy understanding of the structure and contents of the tables. But it should also explain why the tables are needed and structured in a certain way. In order to facilitate the comprehension of the documentation, the documentation itself should be partitioned into a hierarchy of documents each containing a different level of knowledge.

2. Hierarchical Structure. A table is partitioned in groups of data items, each group is subdivided into subgroups, each subgroup is subdivided again, etc. The table is structured in a hierarchical way (Fig. 1). Each of these groups will be called a set of data items or a data element. Hence, the following data elements in their hierarchical order of seven levels (including the file) are presented:

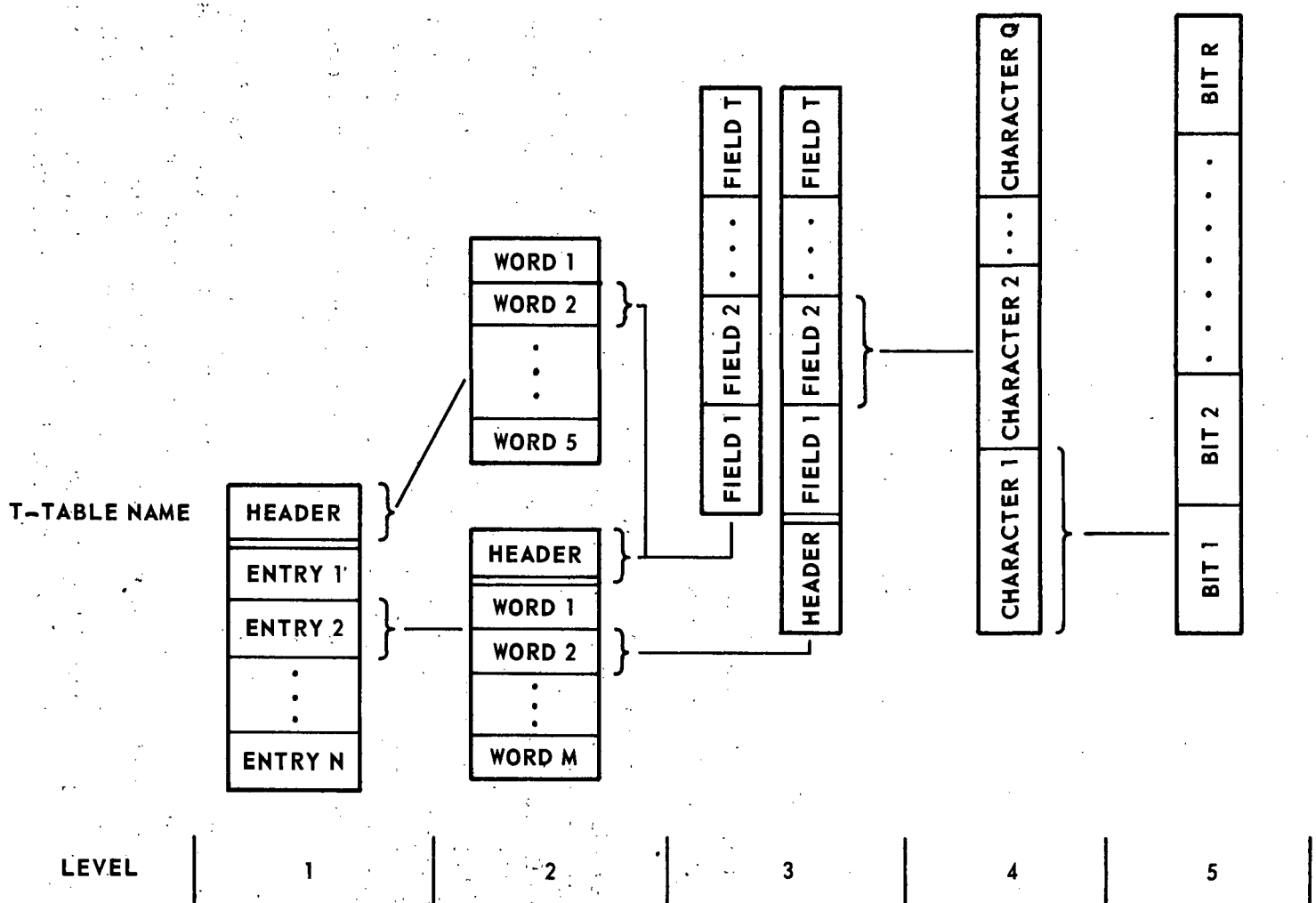


Figure 1. General hierarchical structure of a table.

File	=	Collection of tables
Table	=	Collection of entries
Entry	=	Collection of words
Word	=	Collection of fields
Field	=	Collection of characters
Character	=	Collection of bits
Bit		

All seven levels of a table are not necessary; for instance, it could contain words, characters, and bits only.

To process data being contained in the data elements, one has to locate or identify them uniquely. This is being done by a name that serves as a label or address for a data element. The names should be chosen to express the contents of the data element as accurately and concisely as possible. The name is used to reference a data element and to designate the beginning of its storage space. Each data element covers a certain amount of storage space. In a word-oriented computer, the name of a table is usually identical to the name of the first word of the table. For describing the processing operations, a shorthand notation is used which makes it necessary to distinguish uniquely between various types of references to the data elements of a table.

3. Shorthand Notation for Table References. Each name of a table starts with T- for an easier search of table names. For practical purposes, the name should not be longer than 36 characters. To name a data element of a table, one writes successively the names of the data elements that are connected in the hierarchical structure with the data element to be named. A simple example should clarify this statement. Figure 2 shows a typical table. The names are shown in capital letters to distinguish them from a designation.

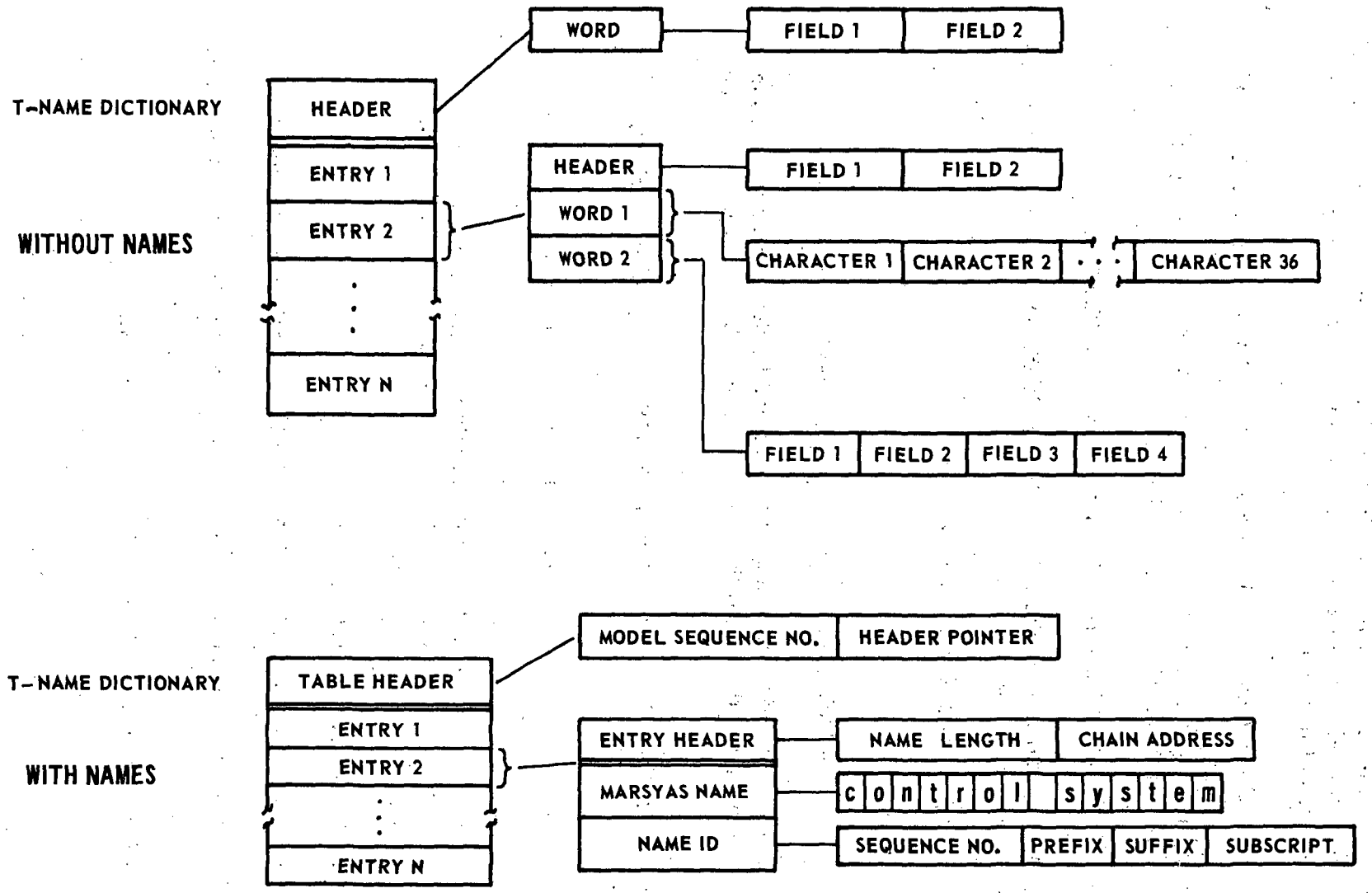


Figure 2. Structure of table T-NAME dictionary in MARSYAS.

For instance, the designation "field 1" means the first field of the field data element, while MODEL SEQUENCE NO. is the name of that field. That name indicates that the field contains the sequence number of the model. The total name or address of that field is

T-NAME DICTIONARY, TABLE HEADER, MODEL SEQUENCE NO.

The total name (or address) of the prefix of the NAME ID of the second entry of table NAME DICTIONARY would be identified by

T-NAME DICTIONARY, ENTRY 2, NAME ID, PREFIX.

Knowing the structure of the table T-NAME DICTIONARY, one could have also identified these two fields by

T-NAME DICTIONARY, header, field 1

and

T-NAME DICTIONARY, entry 2, word 2, field 2.

In general, a data item should be identified by the concatenated names of the data elements in the hierarchical order of the table separated by commas; i.e.,

T-NAME TABLE, NAME ENTRY, NAME WORD, NAME FIELD, NAME CHARACTER

knowing the structure of a table, one can always form a unique composite name for each data item.

If a table contains as entries only a string of characters, the total name contains only the names of two data elements; e.g., NAME TABLE and NAME ENTRY. The name then reads

T-NAME TABLE, NAME ENTRY.

Knowing the structure of the table, NAME ENTRY identifies a character.

The contents of a table are denoted by parentheses around a name. For instance,

(T-NAME TABLE, NAME ENTRY, NAME WORD)

means the contents of the word with the name T-NAME TABLE, NAME ENTRY, NAME WORD. An example is given using Figure 2:

(T-NAME DICTIONARY, ENTRY 2, MARSYAS NAME)

contains the MARSYAS name "CONTROL SYSTEM."

Besides a name, the contents of a data item may also be a specific number; e.g., 35, or any number, or a code for the designation number. We distinguish between these three types of data item contents by the following notation:

Contents = '35'	The contents is the specific number 35.
Contents = 'any number'	The contents can be any real number from $-\infty$ to $+\infty$, integer, fixed point, or floating point.
Contents = 'number'	The contents signals a number; e.g., by code 9999. (No number might be 0000.)

The contents can also be logical combinations of "and" and "or" of these three types of contents, or arithmetic combinations; i.e., +, -, *, /, exp., etc.

The notation of parenthesis can also be used for data element names within a total name (indirect addressing). For instance, in

(T-NAME DICTIONARY (I-ENTRY COUNT), MARSYAS NAME)

the contents of indicator I-ENTRY COUNT is used as the name of the entry in T-NAME DICTIONARY. If the MARSYAS name of the second entry of table T-NAME DICTIONARY should be accessed, (I-ENTRY COUNT) has to be equal to 2. If the location of a data item in one table is dependent on the contents of another data element in another table, one can write, for example,

(T-NAME DICTIONARY, (T-PARAMETER TABLE, ENTRY 5, PARAMETER ID, SEQUENCE NO.), MARSYAS NAME).

This means that the entry name to be accessed in table T-NAME DICTIONARY is given by the sequence number of the parameter ID of the fifth entry in table T-PARAMETER TABLE. One could conceive a nesting of several parentheses and names of data elements.

In searching a table, the case may be that not a specific data item should be identified, but one that has a specific content. Thus, one could ask for any sequence number (i.e., the first encountered in the search) in the name ID of the fifth entry of table T-NAME DICTIONARY which is larger than 100. This would be specified in our notation as

> Any (T-NAME DICTIONARY, ENTRY 5, NAME ID, SEQUENCE NO.) > 100.

If no sequence number greater than 100 could be found in table T-NAME DICTIONARY, this would be indicated by

> No (T-NAME DICTIONARY, ENTRY 5, NAME ID, SEQUENCE NO.) > 100.

In the documentation, the same table and same entry, but different words and fields, might be accessed repeatedly. In this case, one could simply omit the table name and entry name and replace it with a dash, e.g., (T-, NAME ID, PREFIX), and thereby save writing of redundant information.

4. Description of Tables. The documentation of all the tables, including temporary tables, should be separated from the description of the programs. It should show the structure, formats, and contents of the tables. Also, an explanation should be given of why a particular table is needed. The documentation of the tables should comprise the following sections:

a. Purpose of Table (POT).

b. Table Structure Overview (TSO).

c. Table Format and Comments (FAC).

Each section and each table should start with a new page so that it can be ordered differently for the different reader's convenience. Normally, these pages should be ordered alphabetically. To have the pages separately self-contained has the advantage that one can take them out and place them on a big display. Thus, one has them all available immediately when going through the documentation of the programs because all tables have to be overlooked all the time and quickly. Moreover, changes can be documented more easily. Appendix A illustrates an example of the documentation of the MODEL TABLES FILE used in MARSYAS.

a. Purpose of Table (POT). This section explains briefly the major contents of each table and the major usage of the table. If the purpose of the table is not obvious from its contents, the reason for its existence should be given.

b. Table Structure Overview (TSO). The table structure overview should be a graphical representation of each table (on one page only, if possible). The overview should contain all data elements with their names and a list of their possible contents. For variable-length data elements, it should be shown what the length depends on. If a particular data element is used throughout several tables, it may be shown separately. For instance in MARSYAS, the NAME ID is used in different tables.

c. Format and Comments (FAC). This section devotes itself more to the machine-related specifications of each table. It shows the programmer in which format, order, and location in which the table data elements are stored. It also depicts the code of specific words, the exact length, and position of fields and characters.

The comments explain the reasons for the design of the table, its context with other parts of the software, and more general thoughts about the usage of the particular table.

5. Indicators. Indicators in a software system are used to store auxiliary information necessary to control the processing. They can be used by one program only or by many programs. The term "indicator" is generally used for counters, pointers, switches, flags, etc. Usually, they are of fixed length. In addressing them and referencing their contents in shorthand notation, they are treated like tables. Since these indicators do not contain much information, but are much larger in number compared to the tables, each indicator can be briefly documented in the Indicator List by its name, function, contents, and format. The subroutines which communicate with the indicators are listed in the table-subroutine affect diagram, and their names do not have to be repeated in the Indicator List.

In the documentation, the Indicator List is ordered alphabetically (an example is presented in Appendix B). The format for an entry in this table is as follows:

NAME:
(FORTRAN-NAME) } Function and contents; memory space, data type, format.

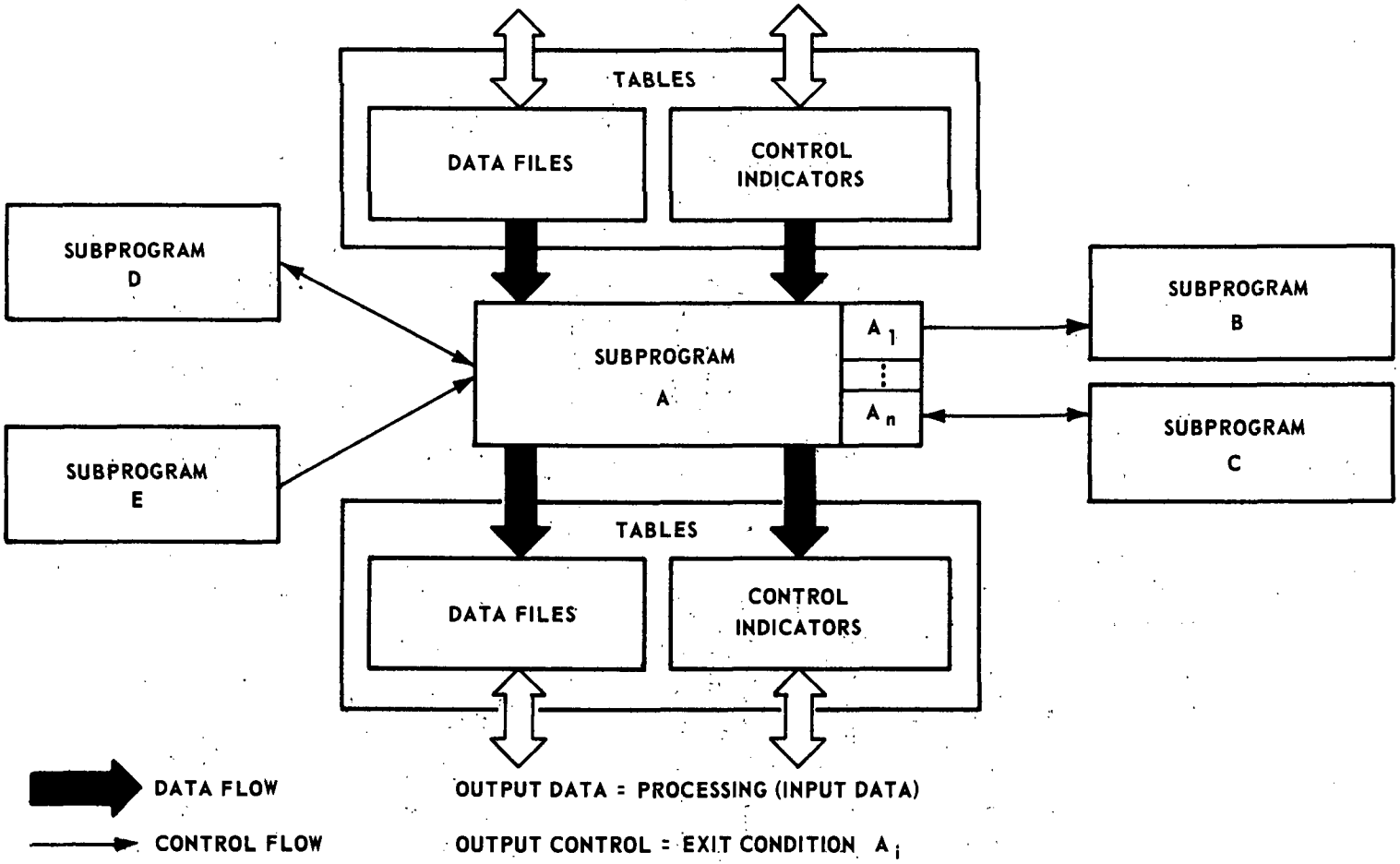
NAME is the unique name used for the indicator in narratives, functional and processing descriptions, and flow charts. There is no restriction on the length of a name, though for practical purposes, it should not exceed 36 characters. To distinguish it from other names, it should start with an I-. The associated FORTRAN name is given in parentheses. The 'function and content' contain a brief summary of the purpose, function, contents, and use of the indicator. The 'memory space' indicates whether the indicator occupies one or more computer words or parts thereof. The 'data type' describes the type of data being stored; e.g., alphanumeric, binary, etc. The 'format' refers to the code and field positions of the data items in an indicator.

6. Buffers. Buffers store information for intermediate use. For instance, when scanning a language statement it is read into an input buffer. A buffer usually occupies several words and is of fixed length. It is treated and documented like an indicator except the name starts with a B- for distinguishing it readily. Usually, the buffer information is used once in the processing and does not control the flow through several subprograms. Therefore, it is less critical for remembering its information.

B. Subprograms

1. Description of Subprograms. The active parts of a software system are its subprograms. They are performing the processing and computing of the data stored in the tables, indicators, and buffers. They read the data from these tables, process them, and store them back to the same and/or other tables. A subprogram may callup some other subprograms and/or it may transfer control to another subprogram without returning to the same subprogram. A subprogram, therefore, is not an isolated entity, rather it is in communication with many passive and active elements; i.e., tables and other subprograms (Fig. 3). In this paragraph, we want to confine ourselves to the actions within a subprogram.

Within a subprogram, we distinguish between two types of actions; one is again passive and the other is active. The passive action means reading only data from a table, indicator, or buffer, comparing it with some other data being read, and deciding what to do next. It acts like a status or condition switch. Analogous to computer hardware, it functions like a logical combinational network which has one input and as many outputs as there are logical states. Graphically, we will represent a logical state by a circle with the number inside (Fig. 4). For instance, when the contents of the indicator I-WORD is examined whether its contents are any 'floating point number' or 'any fixed poing number' or 'any alphanumeric name,' the outcome of this decision has three possible states; i.e.,



NOTE: SUBPROGRAM C IS BEING CALLED BY SUBPROGRAM A WITH RETURN TO A WHILE SUBROUTINE D IS CALLING SUBPROGRAM A. THIS ACTION IS DESIGNATED BY DOUBLE POINTED ARROWS AT THE RIGHT OF BLOCK A AND AT THE LEFT OF BLOCK A, RESPECTIVELY. A SINGLE POINTED ARROW INDICATES THAT AFTER SUBPROGRAM E IS FINISHED SUBPROGRAM A AND B FOLLOW WITHOUT RETURNING TO THE PREVIOUS SUBPROGRAM

Figure 3. Control and data connections of a subprogram.

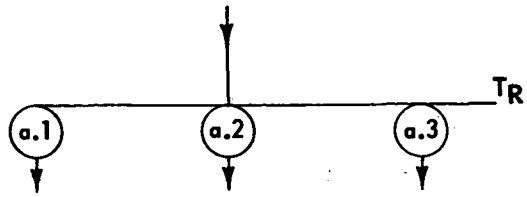


Figure 4. Symbol for logical decision switch. (In this case, there are three states or conditions possible: Ca.1, Ca.2 or Ca.3. "a" is the number of the switch and TR indicates the table from which the information is being read.)

Ca.1 = 'Any floating point number.'

Ca.2 = 'Any fixed point number.'

Ca.3 = 'Any alphanumeric name.'

The decision switch switches the control flow according to these states or conditions.

The active part of an action is the actual processing which reads data from tables, manipulates them, and writes the results into the same and/or other tables. (It is assumed here that the processing does not overwrite other programs or parts of the same program.) It may also take other subprograms (by subroutine calls) to perform this action. A processing element lies always between two condition switches and it can consist of a simple transfer instruction or of the arithmetic expression for a complicated formula. Graphically, we represent the elementary processing action by a fat arrow (Fig. 5). To indicate which subroutines and tables are being used by the processing element P, we add a squared bracket which indicates the set of subroutines $(\overline{S})$ being used, the set of tables $(\overline{T}_R)$ being read from, and set of tables $(\overline{T}_W)$ being written into.

The logical flow can also be directed by a GO TO type instruction in the program which is depicted by a thin arrow. It is assumed that a subprogram has only one entry point. By allowing for a decision switch at the entry point, different entry points can be simulated. The exiting of a subprogram is indicated in the diagram by the exit name E. Figure 6 shows a simple flow diagram of a subprogram described by the symbols of Figures 4 and 5 only.

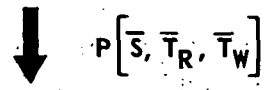


Figure 5. Symbol for a processing element. ($\bar{S}$ is the set of subroutines being used, $\bar{T}_R$ the set of tables read from, and $\bar{T}_W$ the set of tables written into.

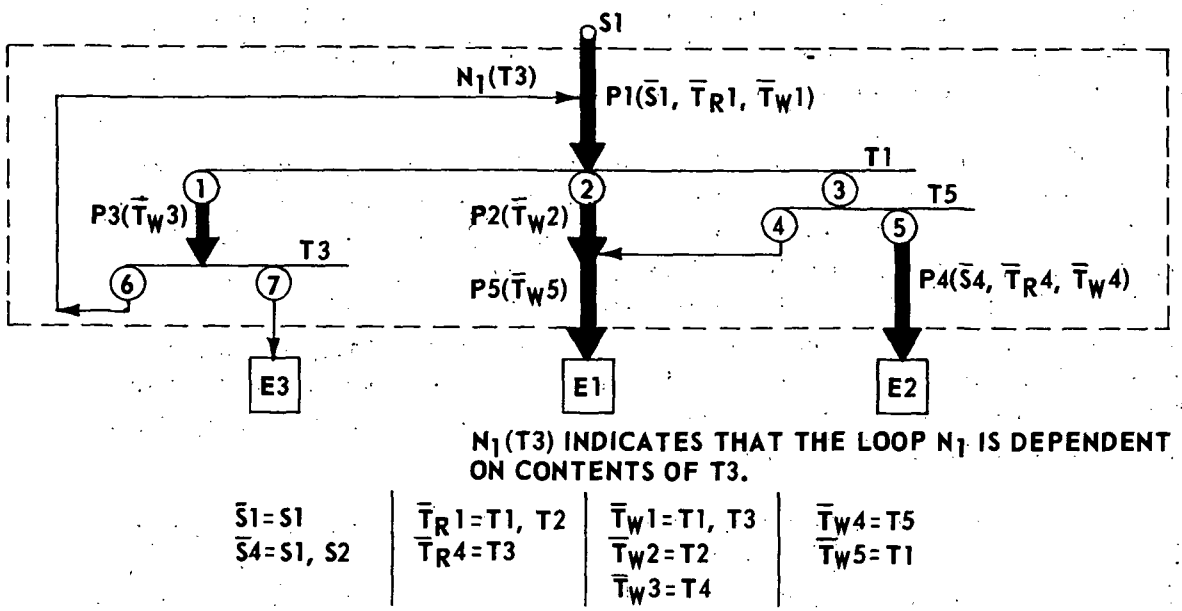


Figure 6. Example of graphical representation of a subprogram S1 with three exits — E1, E2, and E3.

We will discuss later how this graphical representation can be utilized for getting better visibility into software systems.

We will now specify the "primitive operations" that make up a processing element using a shorthand notation. This notation should be machine and language independent, but unambiguous and consistent. These primitive operations only concern themselves with the handling of table contents and simple control functions. They are very easy to memorize and close to the English language. The symbol ← should clearly mark a writing into a table, the symbol = marks a reading and compares a table. These symbols are readily

visible when scanning through the program description. A processing element is labeled by P and a condition by *C to distinguish visually between them.

The "primitive operations" are listed in Tables 1 and 2. These tables can be expanded if necessary. An example in Appendix C should explain the notation in more detail. This notation separates the text from the flow graph. However, the same unambiguous shorthand notation could be applied to the flowcharts as commonly used.

The advantage of the proposed shorthand notation is that it is textual if one chooses meaningful names, and at the same time it is unambiguous. At this detail level of software description, it does not use ambiguous terms like 'access', 'attach', etc. These terms may be used later during the description of groups of processing elements where they are clearly defined by the unambiguous "primitive operations."

The proposed flow graph does not contain text but merely reference numbers to draw large diagrams more easily on a single sheet of paper and to see their topological structure better. Nested loops become more visible. The logical flow through a subprogram can be traced more easily and could result in a higher reliability by having considered all possible branches in the flow. In most cases, the number of tables being used by the processing elements is pretty large and their references would not fit into the diagram. They are omitted in such a case, but can be defined in the Processing/Table Affect Diagram (PTAD), Figure 7.

A certain flow pattern can be observed. Figure 8 depicts a decision tree if no loop is included. For each exit E1...E5, only one path from B to exit E_i is possible and all process elements in that path are in series. One could write a logical equation for each path, where "." denotes a logical "and" for two elements in series. A "+" denotes a logical "or" and constitutes two elements in a parallel path. The equations do not consider the sequence of the processing operations. They are merely a reflection of the static path pattern. This notation could be useful for identifying all paths which have to be tested to verify a complete subprogram. Figure 9 contains two parallel branches which result into four different possible flow paths. If a loop is introduced into Figure 8, parallel paths are added. A bar over the elements in the logical equation means repeated flows through these elements in a loop. It is not the intention to develop a theory in this report rather to suggest that this notation might help to gain insight into the functioning and couplings of software. The relationship between the processing elements and the tables has not been shown yet. Before we go to that we ask the question, "How do we define the performance of a subprogram?"

TABLE 1. LIST OF PRIMITIVE OPERATIONS OF PASSIVE ACTIONS (CONDITION SWITCHES)

Notation	Definition
(ADDRESS 1) = (ADDRESS 2) or term[a]	Comparison between two data elements, i.e., contents of ADDRESS 1 with contents of ADDRESS 2 or any term, yields equality.
≠	Comparison between two data elements, i.e., contents of ADDRESS 1 with contents of ADDRESS 2 or any term, yields inequality.
>	Data element 1 greater than data element 2.
<	Data element 1 smaller than data element 2.
Match in Pa, b =	Comparison between two data elements yields equality during search through one or two tables in processing element Pa, step b.
Match in Pa, b ≠	Comparison between two data elements yields nowhere equality during search through one or two tables in processing element Pa, step b.
End of A	Indication that action; e.g., search through the table
("A" may represent stepping through (T-TABLE))	T-TABLE has ended.

a. Term may be 'any name', 'any number', 'number', '35,' etc., and/or logical and/or arithmetic expression of terms and/or contents of addresses.

TABLE 2. LIST OF PRIMITIVE OPERATIONS OF ACTIVE ACTIONS (PROCESSING)

Notation	Definition
(ADDRESS 1) ← (ADDRESS 2) O or 'term'	The contents of ADDRESS 2 or any term is transferred to memory element ADDRESS 1. A 'term' might be 'any name,' 'any number,' or logical and arithmetic expressions of terms, and/or contents of addresses. The symbol O is present if this transfer is not a one-to-one transfer, but performs an additional operation such as skipping "blanks." It can also be combined with "until," which means that the transfer is stopped when the condition following "until" is reached.
Step through (T-TABLE, ENTRY 1...N)	One entry of the table T-TABLE after another is taken for further processing (comparison, up-dating, etc.) from entry 1 until N.
Step through (T-TABLE, ENTRY 1...N) until (T-ENTRY 1) = (ADDRESS 1) or 'term'	One entry after another is taken from table T-TABLE, starting with entry 1 until the contents of entry 1 equals the contents of ADDRESS 1 or a 'term'.
Call S-SUBROUTINE	Call the subroutine S-SUBROUTINE for execution.
Go to Pa, b or Ca, b	Transfer of control to processing element Pa, step b, or condition switch Ca, b.
RETURN	Return to calling subroutine.
EXIT	Transfer control to next subprogram.

PROCESSING ELEMENTS & EXITS	SUB-ROUTINES		TABLES				
	S1	S2	T1	T2	T3	T4	T5
P1	X		01	0	1		
P2				1			
P3						1	
P4	X	X			0		1
P5			1				
E1	X		ϕ	ϕ	ϕ	1	0
E2	X	X	ϕ	0	ϕ	1	ϕ
E3	X		ϕ	0	ϕ	1	

LEGEND:

0 $\triangleq$ "READING"

1 $\triangleq$ "WRITING"

01 $\triangleq$ "READING FOLLOWED BY WRITING"

ϕ $\triangleq$ "READING" AND "WRITING"

INCLUDES TABLES USED BY CONDITION SWITCHES

INPUT TABLE SET $I = [(T1), (T2), (T5)]$

OUTPUT TABLE SETS $O(E1) = [(T1), (T2), (T3), (T4)]$

$O(E2) = [(T1), (T3), (T4), (T5)]$

$O(E3) = [(T1), (T3), (T4)]$

PROCESSING COMBINATION EQUATIONS

$$P(E1) = \overline{P1 \cdot P3}\ (P2 \cdot P5 + P5)$$

$$P(E2) = \overline{P1 \cdot P3} \cdot P4$$

$$P(E3) = \overline{P1 \cdot P3}$$

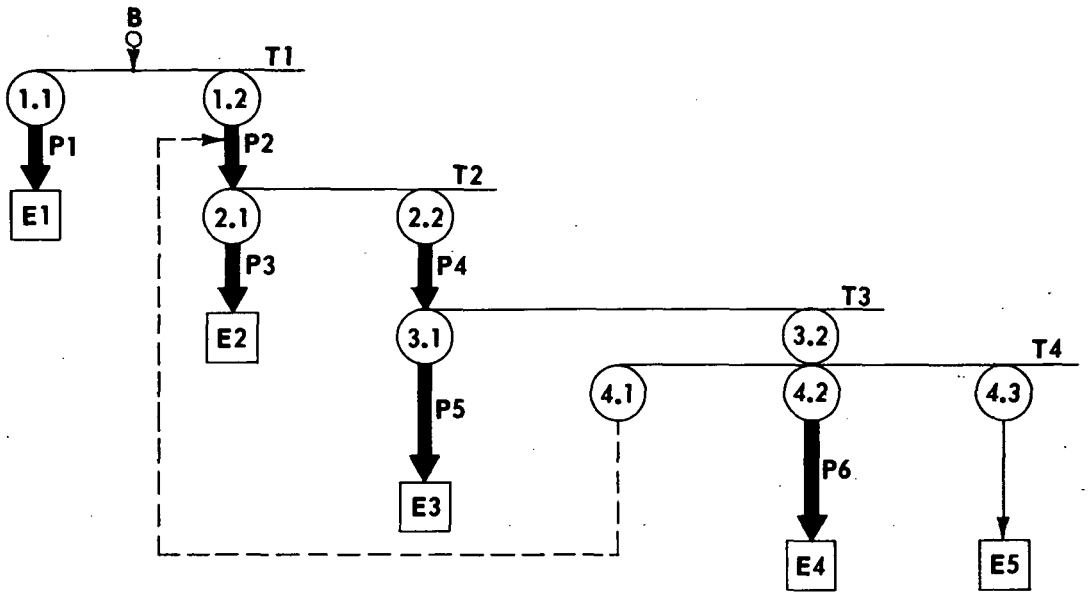
CONDITION COMBINATION EQUATIONS

$$C(E1) = C2 + C3 \cdot C4 + C1 \cdot C6$$

$$C(E2) = C3 \cdot C5 + C1 \cdot C6$$

$$C(E3) = C1 \cdot C7 + C1 \cdot C6$$

Figure 7. Processing/Table Affect Diagram, state sets, and logic equations of example in Figure 6.



WITHOUT LOOP:

$E1 = P1$

$E2 = P2\ P3$

$E3 = P2\ P4\ P5$

$E4 = P2\ P4\ P6$

$E5 = P2\ P4$

WITH LOOP:

$E1 = P1$

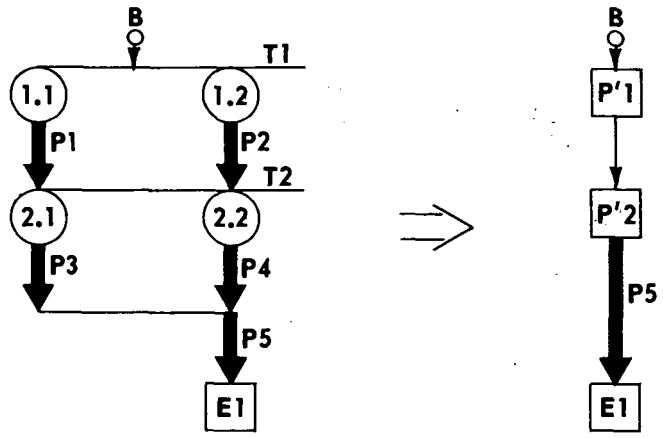
$E2 = P2\ P3 + \overline{P2\ P4}\ P2\ P3$

$E3 = P2\ P4\ P5 + \overline{P2\ P4}\ P5$

$E4 = P2\ P4\ P6 + \overline{P2\ P4}\ P6$

$E5 = P2\ P4 + \overline{P2\ P4}$

Figure 8. Example of a cascaded flow structure within a subprogram.



$$E1 = P1\ P3\ P5 + P1\ P4\ P5 + P2\ P3\ P5 + P2\ P4\ P5$$
$$= (P1 + P2)(P3 + P4)\ P5$$
$$P'1\ P'2\ P5$$

Figure 9. Example of a parallel-series flow structure within a subprogram.

2. Performance of a Subprogram. In this paragraph we try to define the performance of a subprogram in general by knowing only the tables it communicates with, other subroutines it uses, by its processing elements, and by its flow pattern. In this discussion we refer to Figure 6 as a fictitious example. We state first a few definitions.

We call the tables the subprogram is reading from the "input table set" and the tables it is writing into the "output table set." In Figure 6, the "input table set" is [T1, T2, T5]. Each of these tables can contain a particular information which we call the state of the table. Since a table and a set of tables can contain many different information sets, the input table set can assume many input state sets. And for each input state set, only one output state set can be produced. If we assume k input state sets in example Figure 6, we denote them I[(T1), (T2), (T5)] 1...k. Since a subprogram can exit at different points, we associate with each exit an output state set. Hence, Figure 6 shows three output state sets:

$$O(E1) = O_1[(T1), (T2), (T3), (T4)]_{1...k_1}$$

$$O(E2) = O_2[(T1), (T3), (T4), (T5)]_{1...k_2}$$

$$O(E3) = O_3[(T1), (T3), (T4)]_{1...k_3}$$

with the constraint $k = k_1 + k_2 + k_3$.

These three "output state sets" were taken from the PTAD of Figure 7 which shows what tables are being used by the processing elements. It is assumed here that the called subroutines S1 and S2 communicate with the same tables as indicated for P1 and P4.

A subprogram performs properly if it transforms all "valid input state sets" into all required "valid output state sets." By "valid input state set," we mean all those states of the input tables which are used by the subprogram. In our following discussion, let us use the example of Appendix C which describes parts of the subprograms S-SCAN WORD plus the subprograms S-SCAN CHARACTER and S-PLACE WORD. Indicators and buffers are treated like tables.

In S-PLACE WORD, the valid states of I-OPERATOR INDICATOR are 'element mnemonic' and 'no element mnemonic,' because only these two states

are being considered by S-PLACE WORD. In this case, the state 'no element mnemonic' includes all other possible states such as 'connect-operator,' 'disconnect-operator,' 'parameter-operator,' etc., which are not 'element mnemonic.' Other indicators can assume a larger number of states such as I-CHARCOUNT, which can assume any valid value between 0 and 36. If the subprogram is supposed to distinguish between different values of (I-CHARACTER COUNT), e.g., any number > 36 for error check, they have to be defined as a separate valid input state. If the subprogram does not distinguish between the values of the numbers, any number of characters is a valid input state. If no decision is made by the subprogram upon particular information contents of buffer B-WORK AREA, its valid input state is 'any data.' However, for checking out the proper performance of a subprogram, combinations of valid states have to be considered. S-PLACE WORD branches on the condition C2; i.e., if the sum of (I-INPUT AREA POINTER), (I-POINTER MODIFIER), (I-CHARACTER COUNT), and (I-RIGHT PAREN) is > or $\leq$ 73. This means that "typical input states" have to be defined, which are selected input states that exercise all paths of a subprogram. In our case, one set of values of the four above-mentioned indicators has to be chosen which makes the sum greater than 73, another which makes it equal to 73, and a third one which makes the sum smaller than 73. From this simple example, it can be seen that specifying the "valid input table states" requires thorough knowledge of the detail functions of the subprogram. This is especially true when the subprogram uses many input tables, extracts different data from these tables, and performs many decisions on these data. Moreover, it may be necessary to consider the contents of different entries and fields of a table as input table states.

To find the input table set, we look at each table and determine whether it can be read from before it can be written into by the subprogram. If this is true, that table is part of the input table set. A systematical approach is given now.

We again use S-PLACE WORD as an example. First, we set up the PTAD, which simply lists all tables used by the processing elements of the subprogram (Appendix C). A reading is marked by "0" and a writing by 1; the sequence "reading, writing" is marked by 01. To find the input table set we look at one table after the other (i.e., one column after the other). If no 0 exists, we go to the flow diagram and see if a decision switch is dependent on that table. If no switch dependent on that table exists, no zero is entered into the line INPUT TABLE SET of the PTAD. If a switch dependent on that table exists, the 1's in the appropriate column of the PTAD have to

be checked. They tell which P-elements write into the table. We now go back to the flow graph and check all paths if any of the writing P-elements are located before or after the decision switch when we follow the logic flow from the begin B to every exit. If no writing occurs prior to the decision switch, the table is part of the input table set. Otherwise, the table is not part of the input table set. If there is an O in the column of the PTAD, the flow diagram and the 1's of PTAD have to be checked whether a reading occurs before a writing. If so, a 0 is entered in the INPUT TABLE SET line of PTAD. This is done for each column.

The output table set is found simply by checking each table (i.e., each column of the PTAD) for 1's. If there is at least one 1, it is entered into the OUTPUT TABLE SET. This means that a table is considered part of the output table set because it is written into regardless of whether that same table appears in the input table set of another subprogram and its contents is used by that other subprogram.

The tables of the output table set can contain different information, depending on the information in the input table set. One can think that for each input table state set, a corresponding output table state set exists. Without knowing how the subprogram functions internally, the performance of the subprogram could be defined by an input/output table which contains all valid input table states and their corresponding output table states. An example of this input/output table is given in Appendix C for S-SCAN CHARACTER.

One could specify in the design all valid input and output states in the "input/output table" and check the implementation of the subprogram against them. When connecting the subprograms to form a system only the input/output tables of each subprogram come into play. However, generating all valid output table states becomes difficult when the subprogram such as S-SCAN WORD is of larger size. More research has to go into this problem to discover a systematic way of generating the input/output table. It is the intent of this report to instigate research to determine new ways of describing software for gaining more insight into its mechanism rather than to present a final solution.

3. Subdividing a Subprogram. To understand the functions of a subprogram on a somewhat higher level than on the level of the "primitive operations," a subprogram can be subdivided into groups of processing elements and decision switches. One can draw boundaries in the flow graph to minimize the crossing of lines (i.e., interfaces) and to define the function of each encircled area using verbal text. The verbal text should contain references and labels so that its relation to the "primitive operation" is readily visible. In the verbal text, ambiguities otherwise contradiction cannot be avoided, though the designer in writing the text should choose words that as precisely as possible describe the

functions. If the reader of the documentation should encounter an ambiguity, he has the possibility to clarify it by stepping one level down to the description of the processing elements and decision switches.

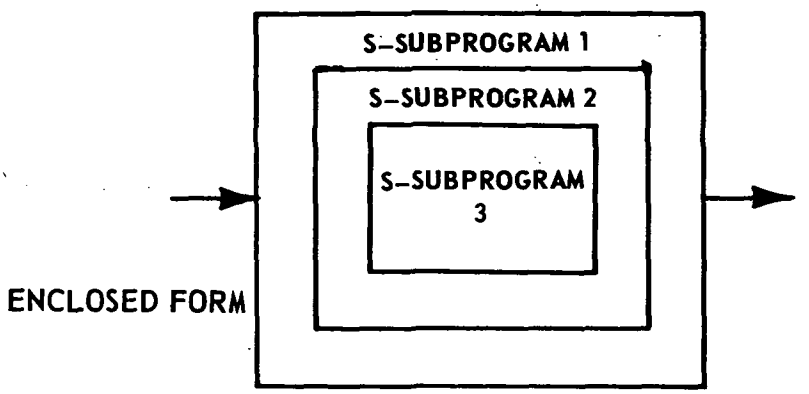
Appendix C.3 defines how a part of S-SCAN WORD is subdivided into the function groups P'1, P'2, and P'3. Their description is included in Appendix D.1. The reduced flow graph is shown in Appendix D.2. The reference of the subroutines which are being called is included in brackets.

To indicate how often the major loops are being executed, the table number responsible for the number of cycles through the loop is attached to each loop in the flow graph. This feature should help to find more quickly those loops of the system which contribute mainly to the computation time.

When designing the software, the size of a subprogram should be limited to such a number of processing elements that the number of function groups is rather small (not more than ten), which can be overviewed readily.

C. Software System

A software system is viewed as a highly coupled logical network of the basic components that were described above; i.e., subprograms, tables, and indicators. The functioning of such a system can be divided into the flow of control through this network and the simultaneous processing of the data stored in the tables. To know which paths the flow will take through the network, the static connections between the subprograms have to be known. These static connections are like roads which connect cities and towns, and the road-map is the collection of all possible connections within a given area. The subprogram connection diagram (SCD) resembles a road map. Two types of connections between two subprograms are possible. One is the subroutine call where the called subprogram returns after finishing its processing to the calling subprogram. The other is a transfer of control from one subprogram to another without return. In MARSYAS, nearly all connections are of the first type which can be depicted graphically as shown in Figure 10. One can foresee several subprograms nested up to many levels. Within a subprogram, the same subroutine can be called at several locations; i.e., by several processing elements. Later, when we wish to trace the flow through the SCD, we would like to specify under which higher-level conditions a subroutine is called. We therefore denote the location of the subroutine call (i.e., the function group number) at the output of the subprogram block in the SCD (Fig. 11).



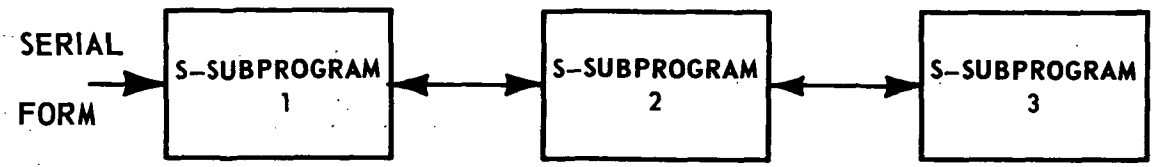


Figure 10. Two types of graphical representation of three nested subprograms.

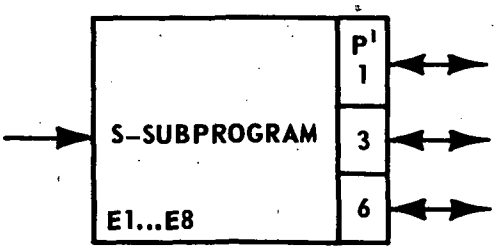


Figure 11. Block used in SCD which denotes the function group numbers P_1' that call other subprograms and the exit point number E.

Since a subprogram can exit at different points, we mark the different exit numbers in the block; e.g., E1...E8. We assume that each subprogram has only one input or begin-point B from which the flow can branch in any way. An example of a system of subprograms which consists of calling subroutines and is nested in three levels is shown in Figure 12.

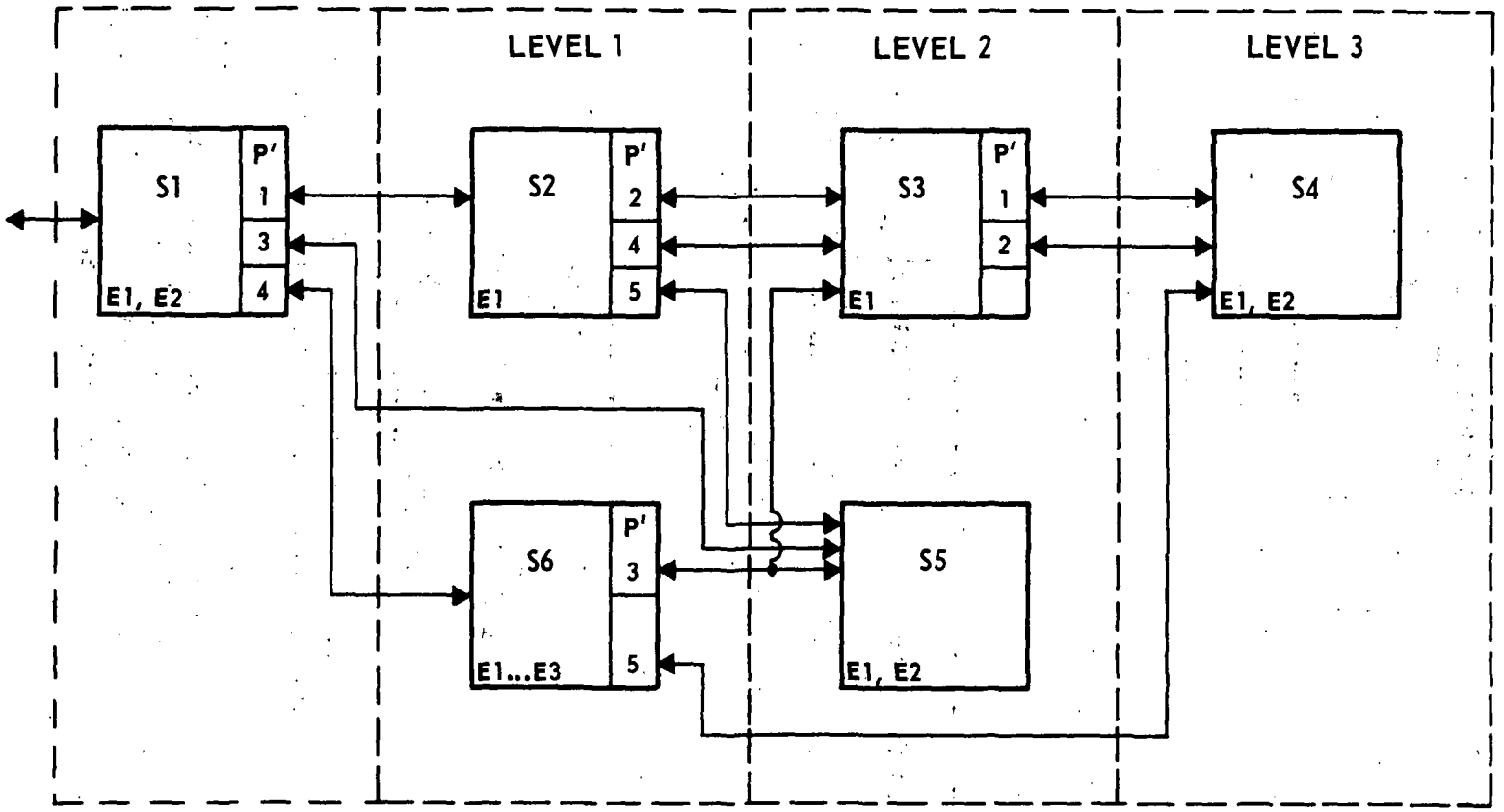


Figure 12. Example of SCD for the program S1 which consists of five called subroutines at three levels of nesting.

One can trace through the SCD for each typical input table state set of program S1. For different groups of input table state sets, different flows exist through the SCD. Within each such group of input table state sets, the flow within the subprograms is different, while at the same time the flow uses the same connection lines between subprograms. To describe and exercise all possible connection paths, the input table state sets have to be chosen properly; which is not an easy task. One can mark the different flow paths by different colors similar to marking travel routes on a road map, but the SCD would become cluttered soon if one draws in all possible flow paths. Therefore, we create another diagram, the subprogram sequence diagram (SSD), which lists all subprograms of a particular flow in sequential order. Loops through subprograms are marked by bars above the subprogram numbers involved (Figs. 13 and 14).

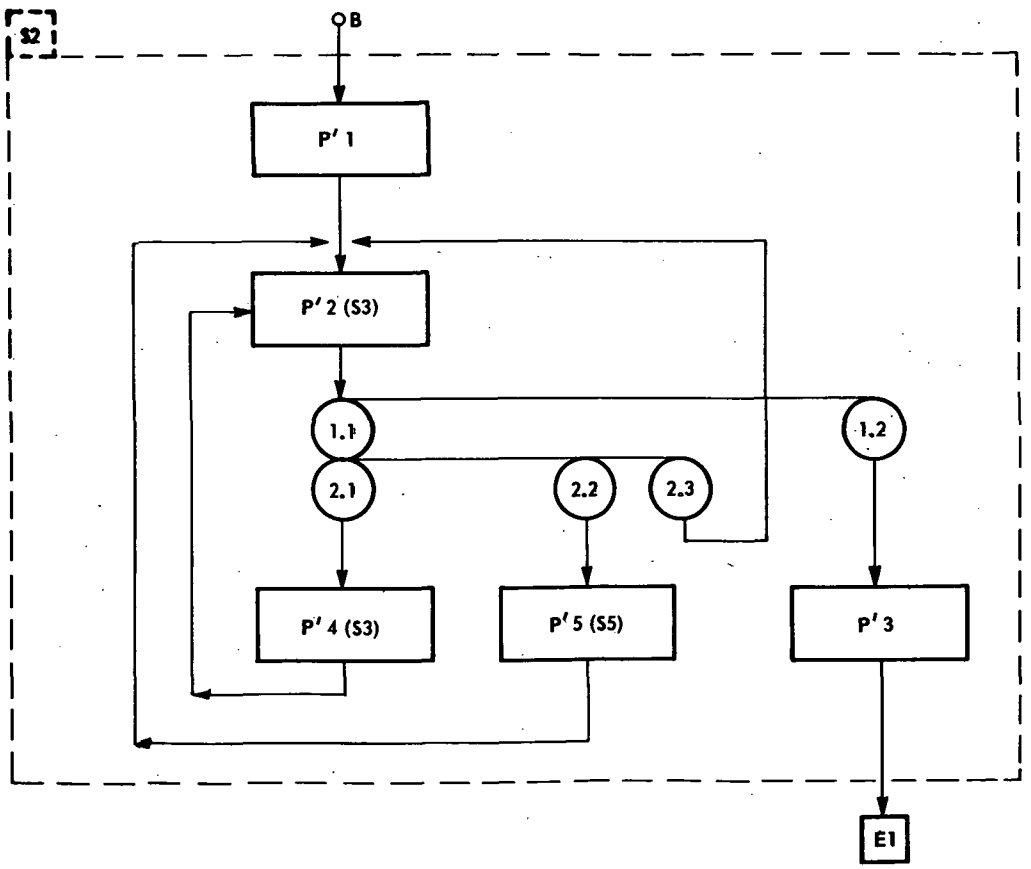


Figure 13. Internal function flow diagram of subprogram S2 of Figure 12, which contains three loops that involve P' 2 (S3), P' 4 (S3), and P' 5 (S5). (This is indicated in Figure 14 by $S2_2$–S3–$S2_4$–S3–$S2_5$–S5. The subscript 2 in S2 is an abbreviation of P'2 of S2.)

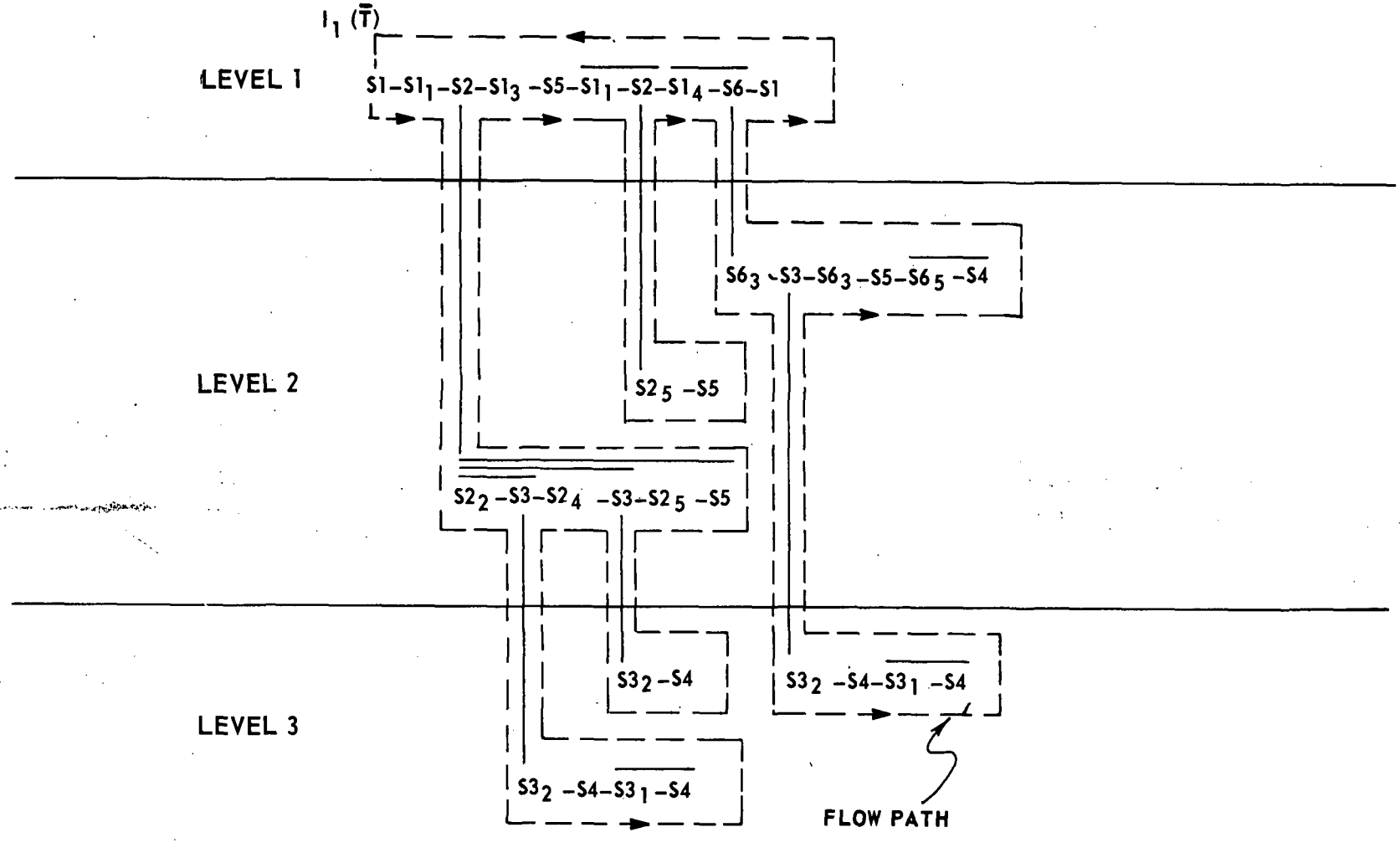


Figure 14. Subprogram sequence diagram (SSD). [Flow path through program S1 of Figure 12 for input table state set $I_1(\overline{T})$. The dotted line is used for explanation only; it shows how the diagram is to be interpreted.]

The (SSD) helps to follow major flows through the software system and to understand its functioning. But it can also be an aid to determine which tables should be overlaid to save core storage. Applying the subprogram/table affect diagram (STAD), we know which tables are being used by each subprogram (Fig. 15). Let us assume subprogram S2 uses the two table sets $\overline{T}10$ and $\overline{T}11$ and subroutine S6 the two table sets $\overline{T}10$ and $\overline{T}12$. Since S2 and S6 are being executed at different points in the flow, the table set $\overline{T}10$, which is common to both subprograms, could be kept in core, while the table set that belongs to the subprogram which is not being executed is transferred to bulk memory. Hence, table set $\overline{T}12$ could be kept in bulk memory while S2 is running and table set $\overline{T}11$ while S6 is running.

SUBPROGRAMS	TABLES T1	T2	T3	T4	T5	
S1	0	1	1			
S2		0		1	ϕ	CHANGE IN S2
S3			ϕ	1	ϕ	AFFECTS S3
S4	1		0			
S5		0	1	0	1	AFFECTS S5
S6	0	1		1	1	
S7	1	0			ϕ	AFFECTS S7

0 READ FROM TABLE 1
1 = WRITE INTO TABLE
ϕ = READ AND WRITE

Figure 15. Example of STAD.

One has to check several major flow paths through the SCD using different input table state sets to ensure that the transfer of tables to bulk memory at different points in the flow is justified.

From the SSD, one can also see which subprograms are being used more frequently and which appear in loops. This can be helpful in determining which subprograms should be refined first for improved computational efficiency; e.g., by writing it in machine-language rather than in FORTRAN. One can draw the SSD in such a way as to show the hierarchical levels of subprograms as they are called. One can see that some subprograms appear in two or more levels of the SSD because they can be called from different levels of the SCD. For instance in Figure 14, the subprogram S5 appears in the first and second level of the SSD. The first level shows the flow through the major subprograms.

D. Proposed Research

As indicated earlier, this report can give only guidelines for an improved description and documentation of software. Research is required in several areas to make software-engineering more a science than an art. Instead of handcrafting programs, software should be designed using mathematically rigorous principles and building blocks. The description and verification of software should then follow proven rules in accordance with these design principles.

In context with this report, the following research tasks are being proposed:

1. Performance of subprograms — A clear definition of the performance of a subprogram, which might contain other nested subprograms, should be established. Rules should be derived on how to measure and verify that performance.

2. Hierarchical processing functions — The description of the processing functions including the tables in a hierarchical order is done presently more or less by narratives. However, a consistent and readable notation should be found which describes unambiguously the higher-level functions in terms of lower-level functions.

3. Methodology for software verification — A methodology for the verification of hierarchically structured software systems should be developed using the primitive operations, flow graphs, and diagrams of this report as a start.

4. Expansion to real-time systems — In this report, no timing and I/O requirements were included. However, for the design and verification of real-time systems software, timing and I/O are very critical factors which have to be considered in any new methodology.

SECTION II. DOCUMENTATION

A. Purpose

The purpose of the documentation of software systems is many-fold. It depends on the various types of personnel who should use it and on the phase of development which it should reflect. Proper documentation of software systems is extremely important because many persons who are dependent on each other are involved in the systems design. Hence, good communication is vital. Moreover, documentation is part of the software end product itself. For certain real-time software systems, the documentation effort has been estimated to be up to 30 percent of the total development costs.

The common purpose of any documentation on software is to inform the reader about the functions of the software to such a detailed level that he can understand these functions and make meaningful decisions on his level.

1. The documentation should serve several categories of personnel; i.e.,

   - User (primarily an engineer).
   - Manager of several software projects.
   - Project manager.
   - Systems programmer.
   - Novice to be trained.

2. The documentation should be useful for different phases, such as:

   - Design.
   - Testing.
   - Verification.
   - Change analysis.

- Training.
- Maintenance.
- Production.

3. The documentation should have certain qualities, which improve
   - Visibility of the design for all phases.
   - Automation of updating the documentation.
   - Automation of retrieving selected information.
   - Generation of the documentation.

Let us briefly state for each category of personnel what information it needs and what functions it performs.

The user has to know how to formulate his problem to the computer. This can be a higher-level, user-oriented language or a specific format of input. He has to become familiar with the error diagnosis which tells about formal errors he made in the input statements. To control the computer program, he has to know all the options for executing the program and for outputting the results.

The technical manager who is supervising software projects cannot have the detail knowledge about the software design that the programmer requires. However, he has to know the principles of the mathematical and computational foundation of the software, the language structure, and the functions of the major components of the software. He should have a good overview of the total software system and its underlying mathematical/computational schemes, and he should know why a particular design has been selected. He should understand the functioning of the software to the subroutine and table level, so that he can make decisions about major design changes, considering trade-offs with regard to computation speed, storage space, manpower, schedule, and money.

The project manager and systems analyst have to know the language, the mathematical and computational foundation, and the software design to a greater detail than the supervising manager. He should also have some familiarity with the code of critical programs of the system, so that he can

make decisions on problems related to the implementation of the software. He should have a good grasp of all facets of the overall system from design through implementation to testing and verification. This requires that the software structure from a higher level to detail level is clearly visible to him so that he can plan and supervise the coding, checkout, and testing of the individual software components and their integration into a total system. He must be able to assess trade-offs in design modifications as part of his responsibility for the efficiency of the software design and implementation. He is also responsible for establishing a development schedule and test plan based on the hierarchical structure of the software system.

The systems programmer should have detail knowledge of the software components and their interplay with each other and the operating system. He has to be familiar with the coding, data formats, and listings of the software during all phases of the development. He designs, codes, and checks out the software, then tests the individual components and the integrated system or major modules of a complex system. He makes detail efficiency trade-offs for modifications in the design and implementation.

The novice to be trained in either personnel category has to be brought on board as quickly as possible. Therefore, he should quickly get an overview of the total system and detail information about the special areas he has to work in. Since the mobility of computer programmers is much higher than compared to other disciplines, the problem of training is an important aspect for documentation.

B. Structure

1. General. As stated in the previous paragraph, the different personnel categories require the description of the software to different levels of detail and with different emphasis. The documentation should reflect these different levels of detail. An overview of all documents is given in Figure 16. Ample references from one level to the next should be made to facilitate the search through the documentation (Fig. 17). Also, the documentation should give reasons why a particular method has been selected when alternative solutions are available. The documents will be used as textbooks as well as reference or handbooks.

First, we begin with the User's Manual which contains the information that the user needs to set up his problem and operate the software system. It explains particularly the language in a readily understandable way by using

	DOCUMENT NAME	CONTENTS
	USER'S MANUAL	PROCEDURES TO SET UP PROBLEMS AND OPERATE SOFTWARE SYSTEM
	MATHEMATICS DOCUMENT	MATHEMATICAL FOUNDATION OF SOFTWARE SYSTEM
	COMPUTATION–ENGINEERING DOCUMENT	CONCEPTUAL DESIGN SPECIFICATIONS OF TOTAL SOFTWARE SYSTEMS
SOFTWARE SPECIFICATIONS	SOFTWARE FUNCTIONAL SPECIFICATIONS DOCUMENT	CHAPTER 1: PROGRAM MODULE SPECIFICATIONS SECTION 1: FUNCTIONAL DESCRIPTION SECTION 2: PROGRAM MODULE CONNECT DIAGRAM (PMCD) SECTION 3: PROGRAM MODULE SEQUENCE DIAGRAM (PMSD) CHAPTER 2: TABLE FUNCTIONAL SPECIFICATIONS SECTION 1: PURPOSE OF TABLE (POT) SECTION 2: TABLE STRUCTURE OVERVIEW (TSO) SECTION 3: FORMAT OF TABLES AND COMMENTS CHAPTER 3: SUBPROGRAM OVERVIEW SPECIFICATIONS SECTION 1: PURPOSE OF SUBPROGRAMS (POS) SECTION 2: SUBPROGRAM CONNECTION DIAGRAM (SCD) SECTION 3: SUBPROGRAM/TABLE AFFECT DIAGRAM (STAD) SECTION 4: SUBPROGRAM SEQUENCE DIAGRAM (SSD) SECTION 5: SUBPROGRAM CHANGE AFFECT DIAGRAM (SCAD) CHAPTER 4: SUBPROGRAM FUNCTIONAL SPECIFICATIONS SECTION 1: DESCRIPTION OF FUNCTIONS SECTION 2: FLOW GRAPH B SECTION 3: INPUT/OUTPUT TABLE SETS
	SOFTWARE PROGRAMMING SPECIFICATIONS DOCUMENT	SECTION 1: PROCESSING DESCRIPTION ("PRIMITIVE OPERATIONS") SECTION 2: COMMENTS SECTION 3: FLOW GRAPH A SECTION 4: PROCESSING/TABLE AFFECT DIAGRAM (PTAD) SECTION 5: ERROR DIAGNOSTICS SECTION 6: TABLE DETAIL FORMAT SECTION 7: INTERFACES WITH OPERATING SYSTEM
	LISTINGS	PROGRAM CODE (+ COMMENTS) IN FORTRAN

Figure 16. Overview of software documents.

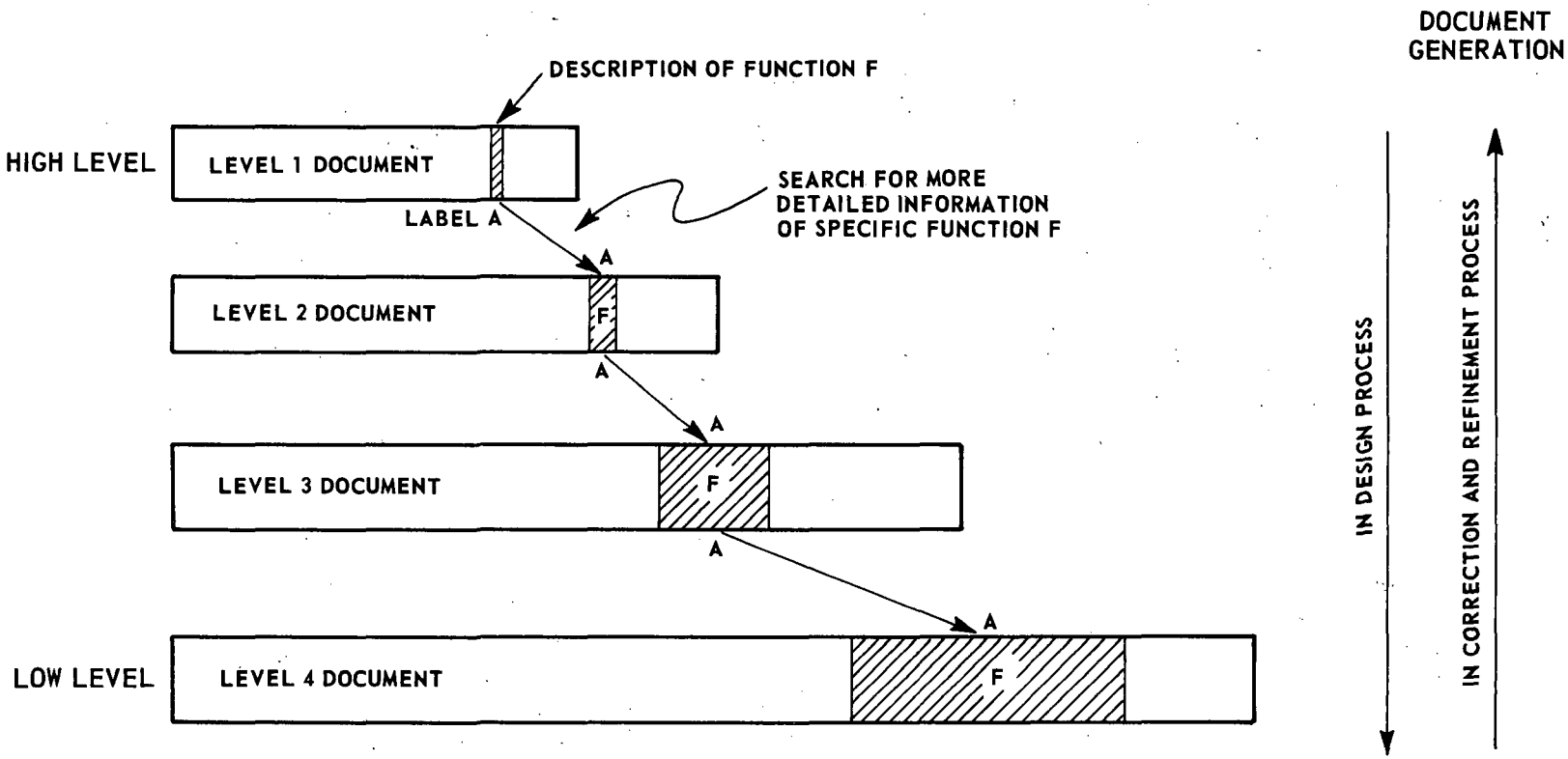


Figure 17. Hierarchy of documentation.

engineering terms and examples. It should also contain operating procedures such as setting up control cards or writing control statements on a teletypewriter. It acts as a tutorial and training manual. Although the User's Manual is probably the first document that a novice will read to learn about the software system, it is generated after the software system has been implemented and tested so that it reflects the latest modifications [4].

Note: The examples referenced are MARSYAS documents, which in their structure and contents are indicative of what is being strived for; however, they do not always comply with the standards set by this report.

2. Language Specifications Document. For the software designer and implementer, a more rigorous Language Specification Document [5] has to be generated. It describes the user-oriented language in mathematics-like notation which does not contain ambiguities or inconsistencies. This document explains the structure of the language and the meaning and format of the statements (syntax and semantics). Besides the Mathematics Report, it is the foundation of the software system.

3. Mathematics Document. The Mathematics Document [6] describes the mathematical foundation of the software system. It contains the analytical equations and the various mathematical steps which have to be taken to arrive at the mathematical solution. The algorithms and formulae to solve these analytical equations are then shown and explained. Special conditions and constraints, such as the bounds within which the mathematical methods are valid, should be given. Analysis of numerical errors such as truncation, round-off, and propagation errors should be included. If available, results of numerical experimentation should be included.

4. Computation-Engineering Document. The Computation-Engineering Document should provide the link between the Mathematics Document and the various software documents. It describes the principles of the computational schemes used and depicts major block diagrams and flowcharts to show the logic flow of the major actions. This document should explain each functional step such as a procedure is explained. The description of these steps has to be of such detail than one can exercise the functions of the design step-by-step manually on paper. One should make use of graphs, diagrams, and narratives — whatever is the best means to expound the design. References to the Mathematics Document and Software Functional Specifications Document should be made whenever possible. For illustration, it should contain examples. This document should serve as a basis for the software design conveying the concept of the total design.

5. Software Specifications Documentation

a. General. The software specifications documentation should be written in several levels so that the different personnel categories mentioned above can obtain the depth of knowledge required. However, these documents have to be linked together through extensive use of references and labels. One can think of the software specification documentation as a hierarchy of various documents. Each higher-level document in the hierarchy builds on the one underneath it (Fig. 17). The lowest-level document contains the listings of the programs which are a precise description of the programs to be executed in the machine. It would be ideal if one could build the document of the next higher level in precise, unique correspondence to the listings and, at the same time, reduce the detailed information without losing the meaning of the information. However, at the present time, the process of generating the next higher-level document requires human judgement which will introduce unavoidable ambiguities. Also, during the design process, the documentation is generated from the higher level to the lower level until the coding (reflected in the listings) is generated. Because of corrections in the code during the development, corrections of the higher-level documents have to be made. The documentation has to go through several iterations through all levels, particularly the lower ones, until it reaches a version which can be officially released. To go from one level to another, ample references and labels should be used. Thus, the description of a function or a group of functions can be defined by several functions of the next lower level. Through the labels one can always obtain more detail information from the next lower level in case of questions because of ambiguities in the higher-level document.

The human language also uses expressions of different hierarchical levels which are often of ambiguous meaning. This ambiguity can be reduced or even removed if one defines the higher-level expression by several lower-level expressions. For example, the expression "to process the Tables A and B" can be represented by several detail or primitive operations (Fig. 18). Hence, in the documents of lower detail level, human language expressions of a lower level will prevail over those of higher-level expressions of higher level.

The description of software can be accomplished in different forms; i.e., in form of narratives, of short-hand notation or symbolic, and of graphic diagrams and charts. We will use all forms whenever best suitable.

We will briefly describe the various software specifications documents in their hierarchical order.

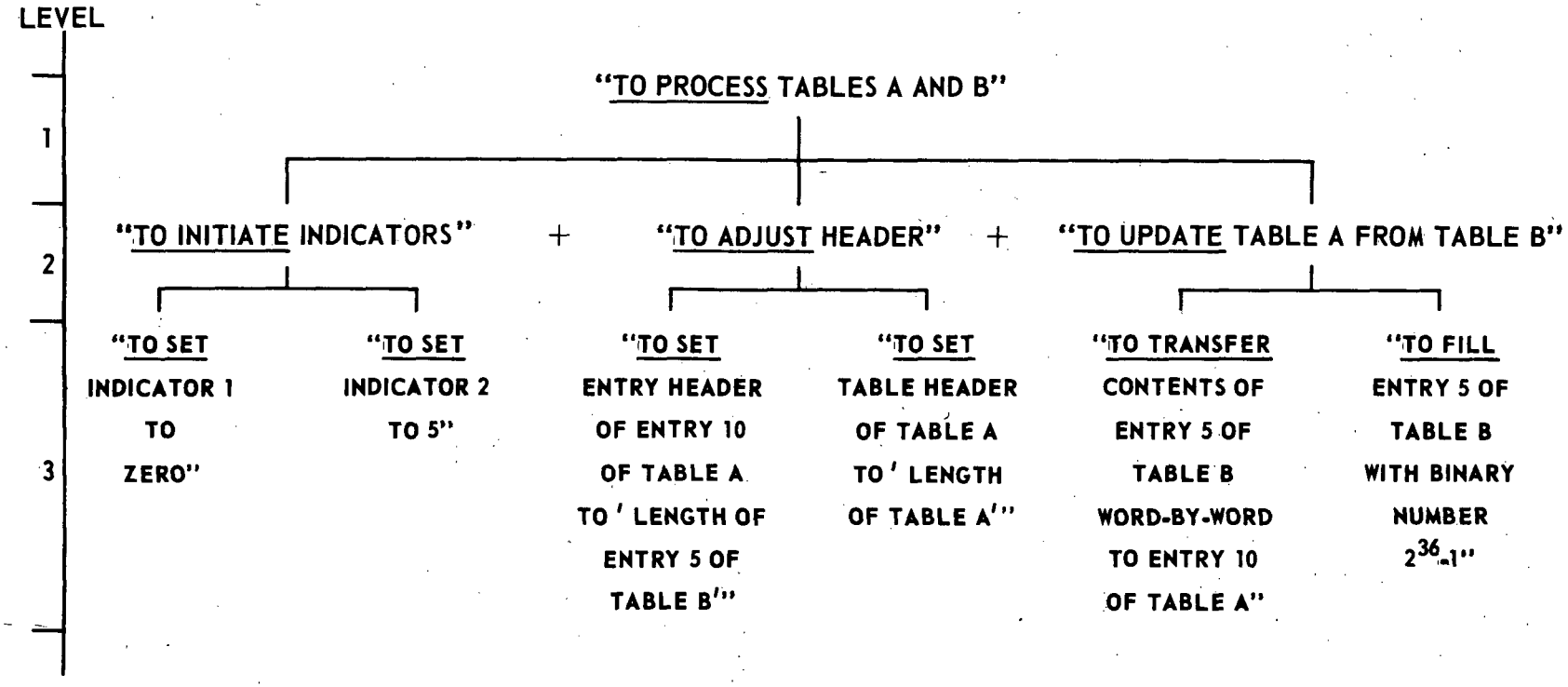


Figure 18. Example of hierarchical human language expressions.

b. Software Functional Specifications Document. The Software Functional Specifications Document is divided into several chapters. Chapter 1 (Program Module Specifications) describes the purpose and functions of each program module and major tables or files used by these program modules. A program module is a collection of programs which performs major functions. It represents the first subdivision of a software system; e.g., the simulation system MARSYAS contains six program modules. Chapter 1 also contains the Program Module Connection Diagram (PMCD), which depicts all possible interconnections between the program modules, and the Program Module Sequence Diagrams (PMSD), which lists all possible flow sequences along the interconnections of the program modules.

Chapter 2 (Table Functional Specifications), in separate sections, delineates the purpose and data contents of each table, indicator, and buffer. (See Part 1, 1.4 for detail.) Examples are given in Appendices A and B.

Chapter 3 (Subprogram Overview Specification) describes the purpose of each individual subprogram grouped by Program Module and outlines briefly the major processing functions together with the major tables being used. The description should not exceed about one-half a page per subprogram at the average. This chapter includes also the SCD, which depicts all possible interconnections between the subprograms (see Section I.C of this report). The Subprogram/Table Affect Diagram (STAD), the SSD, and the Subprogram Change Affect Diagram (SCAD) are also part of this section (see Section I.C of this report). The STAD shows tables, indicators, and buffers that are being used by the subprograms. The SSD portrays the major flow sequences through the SCD. The SCAD illustrates which other subprograms are affected by a change in a subprogram. (An example is given in Appendix E.)

Chapter 4 (Subprogram Functional Specifications) goes into more depth than the overview specifications of Chapter 2 and is based on the Software Programming Specifications Document. The first section (Description of Functions) describes a functional group of processing elements and decision points in a condensed textual form for each subprogram. These functional groups are well defined by encircling a coherent area of the detail flow graph A, which is part of the programming specifications. The second section contains the flow graph B which is a topographical representation of the subprogram at a higher level than that in the programming specifications. The same labels as used in the programming specifications allow easy referencing to the more-detailed specifications. The third section contains the input table set and output table set. (Also see Part 1.2.2.) An example is presented in Appendix D.

c. Software Programming Specifications Document. The next lower level document (just above the program listings) is the Software Programming Specifications Document, which describes in its Section 1 the various subprograms by primitive operation in shorthand notation. From these specifications, programs in any computer language, e.g., FORTRAN or Assembly Language, can be written. This section should also contain a glossary of the shorthand notation so that it can be expanded any time if necessary. Section 2 gives comments to the description of the primitive operations. Since the primitive operations contain the essential information about the actual steps within the computer program, the comments contain explanatory information as narratives. The comments also explain why certain functions are being performed and in which context with other functions. They should be labeled and located in such a way that they can be read together with the primitive operations of Section 1. In Section 3, the detail logic flow graph A is depicted. It illustrates the logical structure of a subprogram by its processing elements and decision points. Section 4 constitutes the PTAD which demonstrates the relationship between the processing elements of each subprogram and the tables they write into and read from. Also, from this diagram the input table set and output table set can be derived. Section 5 describes the error diagnostics exits of each subprogram. In Section 6, the detail format of the table should be documented if it is not feasible to include the format already in Chapter 3 of the Software Functional Specifications Document. (Also see Part 1.1.4.) An example is given in Appendix C.

The labels used in Section 1 should be used in all other sections for easy reference. Section 7 describes the interfaces of the software system with the operating system. This includes the assign and control statements, files, subroutines, and procedures of the operating system used by the subprograms.

d. Program Listings. The program listings are a printout of the actual computer code in FORTRAN or assembly language with comments further explaining the code. These comments should also contain labels used in the Software Programming Specifications Document to reference readily pieces of code to the "primitive operations" and from there to the functions in the Software Functional Specifications Document. This is important when changes in the design or code are made so that all related parts of the documentation are updated.

C. Updating of Documentation

The software documentation consists of may parts that are related to each other. This means that a change made in any one document may result

in changes in other documents and/or in other chapters or sections of the same document. The more related parts that exist, the bigger the update job becomes.

The updating can be divided into two tasks: (1) the places in the documentation have to be found where to update; i.e., one has to identify all locations that are related to a change, and (2) the information at these places has to be changed.

To facilitate mainly (1) above, certain rules in uniting the documentation should be observed:

1. Within each chapter and/or section, the text should be grouped in a systematic way according to the items being described, such as program modules, subprograms, tables, etc. Within each group, the items should be documented in alphabetical order of their names.

2. Each item should start with a new page.

3. The item name and date should appear on the head of each page.

4. Consecutive page numbers should be used within the description of each item.

5. Use labels as much as possible.

These rules also allow different users to compose the documentation in a different order suitable to them.

The STAD's can also be used for cross referencing to locate the places where updates are required.

To reduce the graphic arts work, it is suggested to simplify the flow graphs so that they contain only referencing labels instead of text.

In the process of the development of a complex software system, the documentation has to be modified in several iterations. First, the designer starts out with a Language Specifications Document after he received the requirements from the potential user. He then, or simultaneously with the language, formulates the mathematical foundation in the Mathematics Document. From those two documents, he developes the Computation-Engineering Document, which is the basis for the Software Functional Specifications Document,

which describes the software design on a functional level. From this document and knowing the capabilities of the computer operating system, he can generate the Software Programming Specifications Document which defines the logical detail of the various elements of the software and the total integrated system. This final document then allows him to generate the program. Each step in the development process will require a revision of the next higher level documents since at each step, which leads to a more detailed design, changes have to be expected. For instance, when the programming specifications are being generated, the designer may find that certain processing elements are repetitious according to the functional specifications. He would then combine these processing elements and may define a new subroutine. These changes would then have to be reflected in the functional specifications. Hence, the process will iterate several times between design steps, more frequently between adjacent steps (Fig. 19). When the design is finalized, all labels and reverences have to be checked. Also, one has to ensure that the description of the function groups in the functional specifications are congruent with the processing elements in the programming specifications.

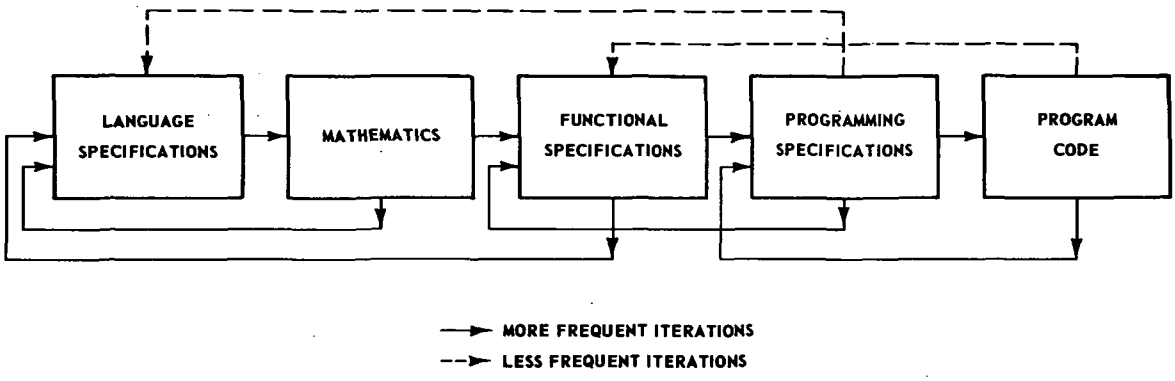


Figure 19. Major steps in software development.

D. Verification of Software

The verification of software is a difficult task and no general method is available which could be applied to prove that all parts of the software and the integrated system are working properly. Much research is required to derive generally applicable methods which verify complex software systems in a systematical way. Therefore, in this report, only a few comments can be made as to how the proposed approach of software description can help the software testing and verification.

It is assumed that the software is of modular design; i.e., it consists of a hierarchy of subprograms and tables.

First, we try to define the process verification in the same way as we do for a hardware system. Verification is the process of comparing the measured performance of an actual system with the performance specified in the design document. Usually, a complete hardware system is verified by first testing the performance of the individual units, then that of functional groups or subassemblies, and finally that of the total system. The performance of a piece of hardware (e.g., a linear electronic feedback amplifier) can be clearly specified by a few parameters and functions regardless of the detail implementation of the hardware design. The power supply of the amplifier is described by the main voltage and wattage. The dynamic characteristics of the amplifier are described by the frequency gain and phase responses. The input and output impedances are the interface characteristics; i.e., they determine the coupling effect on the system when the amplifier is connected to other units in the system. Tolerances and environmental operating conditions can be specified by numbers. However, the performance of a subprogram cannot be specified as clearly as that of a piece of hardware. As is indicated in Section I.B.2 of this report, there are so many output table state sets to the input table state sets that typical state sets have to be selected. The definition of these typical state sets is not a simple task. For this reason, in praxis the design of a subprogram is more or less specified by its implementation; i.e., by its processing elements, decision points, and tables.

The checkout and verification of a complete software system is performed in several steps. The lowest level subprograms in the hierarchy are checked out first, then those of the next higher level. Then a group of subprograms that perform a major function is checked out. Thereafter, these groups are put together to form the system. Then, the interplay of these groups in a system is checked out.

In the checkout and verification of a subprogram, two basic steps can be distinguished. First, all possible paths through the tree-type flow graph (including loops) which depend on the various states of the decision switches are checked for proper flow. (We assume here, that those switches can change their state during the execution of the subprogram; however, they will not be deleted or modified during the execution.) Secondly, the processing of the processing elements using the tables is checked by testing the contents of the tables. For this purpose, the typical input table state sets and their associated output tables state sets at critical points in the flow graph should be specified from the functional specifications. Then the subprogram is

executed with these input table state sets as initial conditions until it reaches the specified points in the flow graph. The contents of the affected tables are compared to the specified output table state sets. If they match, the particular input table state set/output table state set (which might be called the transfer state set) and involved processing elements are checked out. This is done for each typical input table state set.

After all subprograms, or groups of subprograms, of the same hierarchical level (starting with the lowest level) have been verified, the subprograms of the next higher level are verified in the same way until the complete system is verified. The typical input table state sets for the complete system can be generated by using typical test cases of input data. The process of verification was described here rather coarsely. However, it should be pointed out that the various diagrams and notations used in the description of the software are very helpful for a more systematic approach of software verification. For instance, the PTAD can be used to set up the input table sets and output table sets. The Table Functional Specifications indicate the possible states of the tables. The primitive operations tell quickly which operations are performed and which parts of the tables are used. The SCADs tell which subprograms affect other subprograms. The application of the diagrams can be more refined and it might be possible to generate rules on how to verify a complex software system in various well-defined steps. Instead of verifying the total system whenever a change is made, only those portions which are affected have to be verified. This reduces the verification process considerably.

The tables and diagrams should help to establish a test plan for the checkout and verification of the software. This plan should show the sequence in which the various components (from the lower level to the higher level) are tested. It should also contain the various table sets and state sets used in the test process.

If the software system is large, an overlay diagram should also be generated. This diagram should depict graphically which tables are overlaid in core during major flows through the SCD. During the verification, the overlay structure has to be tested to assure that no data needed for processing are destroyed.

E. Automating the Documentation

Since the documentation of a complex software system and its updating becomes a horrendous effort, the use of the computer should be contemplated for various phases in the documentation process. The computer could be used for the following major operations:

1. Storage of documentation.

2. Extraction of essential information.

3. Selective retrieval of information.

The various documents could be stored in the computer and when changes have to be made, the computer could perform updating including format editing and provide a clear printout quickly. However, the print of the documents would be restricted to capital letters and standard key-punch symbols. The tradeoff in economics between the manual update and the computerized one should be performed first before one decides on how far in the automation one should go. A program for automatic documenting is available on the UNIVAC 1108 and it is used for documenting the UNIVAC 1108 operating system.

The computer could perform the tedious operation of extracting information from various types of documents. For instance, if Section 1 of the Software Specifications Document (subprogram description using primitive operations) is stored in the computer, most diagrams and flow graphs could be generated automatically from it, and requires extracting the names of those tables that are written into and read from. The alphabetical ordering of the names is also a much easier task for the computer than for the human being. The same information could be extracted from the listings, either by a special program or by the compiler. However, if indirect addressing is used in a primitive operation or statement, it becomes difficult to find the possible contents of the variable with the indirect address. In that case, the Table Specifications Document is needed which specifies all possible contents of a table, indicator, and buffer. From the call statements, the SCD can be generated.

Once certain documents are stored in the computer, specific portions of a document can be searched for and retrieved much faster than by hand. For instance, if a subroutine is being changed, the programmer could ask the computer to print or display on a CRT the names of those subprograms that are affected by this change through the SCAD. The computer could also generate cross references between any names or labels which could be presented to the user. For instance, the user might want to find out quickly by which subprograms and decision switches a particular indicator is used. This capability might be especially useful if a complex software system is being developed by many programmers who are at different locations. If the names of the programmers working on different subprograms are stored in the computer, the

names of those programmers affected by a change could be notified immediately. Also, the impact of a proposed change on the total software could be assessed more quickly by obtaining the names of subprograms and tables being affected.

Presently used data management systems which have a more generalized structure such as MSFC's Management Information and Display System (MIRADS) could be used for this purpose (Fig. 20) [10].

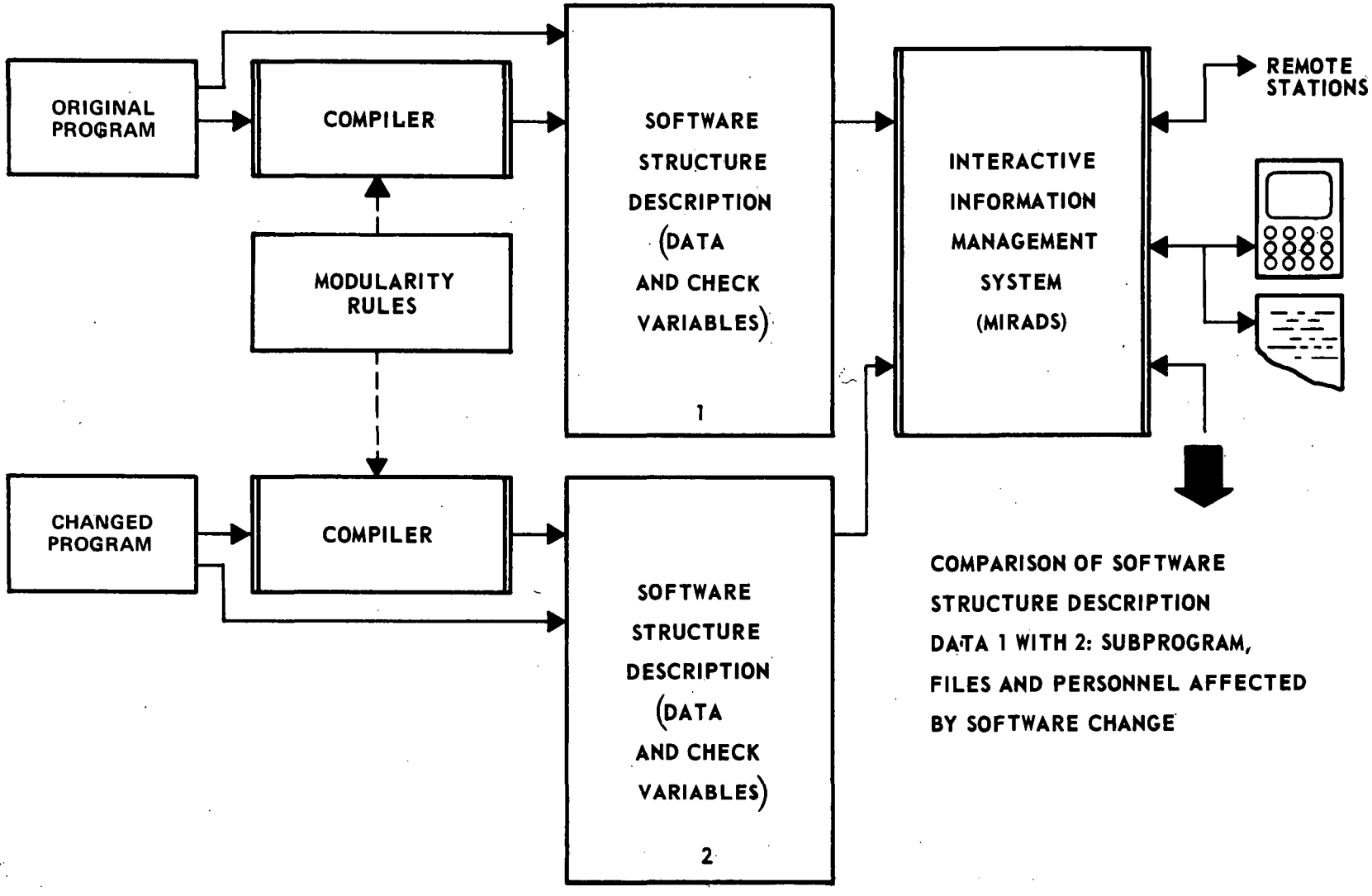


Figure 20. Computerized quick-look software change impact analysis.

APPENDIX A. The "Table Functional Specifications" of the MODEL TABLES FILE of the Software System MARSYAS[1]

The tables used in the DESCRIPTION MODULE of MARSYAS were selected to illustrate how tables should be documented. This selection of tables comprises the MODEL TABLES FILE, which contains the output information of the DESCRIPTION PROGRAM MODULE. This program module translates the statements of a MARSYAS program into these tables. The SIMULATION PROGRAM MODULE then takes the information from these tables and generates the object program which is a FORTRAN program ready for subsequent compilation and execution [8,9].

1. These appendices are extracts of Functional and Programming Specifications from DESCRIPTION Program Module of MARSYAS as an example. The specifications presented in these appendices are not of the latest MARSYAS version. They are taken from the MARSYAS Programming Specifications, October 1968.

1. Purpose of Tables (POT)

a. MODEL TABLES FILE. The MODEL TABLES FILE contains in compact form all the information which is written in the DESCRIPTION MODULE of a MARSYAS program describing a model. These tables contain an image of the block diagram of a model and serve as intermediate tables for the SIMULATION PROGRAM. The DESCRIPTION PROGRAM MODULE scans the statements of a MARSYAS program and fills the MODEL TABLES FILE in such a form and with such additional bookkeeping information that the SIMULATION PROGRAM MODULE can generate the FORTRAN object program file. The MODEL TABLES FILE is necessary since the statements of a MARSYAS program can be written in any order and the CONNECT statements for the same model can be written differently using different connection paths. Therefore, the MARSYAS statements can be processed by the SIMULATION PROGRAM MODULE only after the MODEL TABLES FILE has been created.

b. T-CONNECTION TABLE. The CONNECTION TABLE is a list of terminal pairs which are connected. Each pair consists of a predecessor, which is the terminal transmitting the signal flow, and a successor, which is the terminal receiving the flow. Each terminal is uniquely identified by its NAME ID.

c. T-ELEMENTS TABLE. The ELEMENTS TABLE is a list of NAME IDs corresponding to the elements contained in the model. (It also contains the ID of any submodels which are contained in the model.) It contains information about the type of each element and it points to submodels.

d. T-MODEL INPUT TABLE. The MODEL INPUT TABLE is a list of the name IDs which correspond to the input terminals of the model. Usually, these IDs will agree in their sequence number and attribute fields. Their subscript fields (terminal numbers) will go from 1 to the final value, ascending by 1. However, if a model has been MODIFIED, it is possible that terminals have been deleted, etc. This necessitates this table, which identifies each model input terminal uniquely.

e. T-MODEL OUTPUT TABLE. The purpose of this table is analogous to that of the MODEL INPUT TABLE, except the information in this table is related to the model output terminals instead of the model input terminals.

f. T-NAME DICTIONARY. The NAME DICTIONARY is a table which provides the MARSYAS processor with the capability of translating user MARSYAS names into their appropriate NAME ID words, and vice versa. An entry in the dictionary consists of an alphanumeric name, immediately followed by its NAME ID.

g. NAME ID. Each user-assigned MARSYAS name, which can be up to 36 characters long, is given one unique compact UNIVAC 1108 computer word (36 bit), NAME ID. In the internal operation of the MARSYAS processor (in the various model tables, etc.), this ID is used instead of the alphanumeric name. The NAME ID gives information about the object it refers to and is of uniform length, thus making processing easier and conserving storage.

h. T-PARAMETER TABLE. The PARAMETER TABLE is a list of the parameter values of each parameterized element of the model. An element which does not have any parameters has no corresponding entry in the PARAMETER TABLE. Those parameters which are given by their name and their numerical value in a PARAMETER STATEMENT are identified as named parameters.

General. For each table, one page is used to depict the structure and contents of a table. For use with the subprogram description, these pages may be assembled on one sheet and reduced in size so that they can be viewed immediately. These tables are UNIVAC 1108 word oriented; i.e., one line of names represents one or a multiple integer of words. For instance, in T-NAME DICTIONARY, the header consists of one UNIVAC 1108 word with the two fields MODEL SEQUENCE NO. and HEADER POINTER. First, the collection of all model tables; i.e., the MODEL TABLES FILE is presented.

MODEL TABLES FILE[2]

T-NAME DICTIONARY	1 N_1
T-MODEL INPUT TABLE	1 N_2
T-MODEL OUTPUT TABLE	1 N_3
T-ELEMENTS TABLE	1 N_4
T-PARAMETER TABLE	1 N_5
T-CONNECTION TABLE	1 N_6

Variable Number of Entries

N_1 = Number of MARSYAS names in model.

N_2 = Number of model inputs.

N_3 = Number of model outputs.

N_4 = Number of elements.

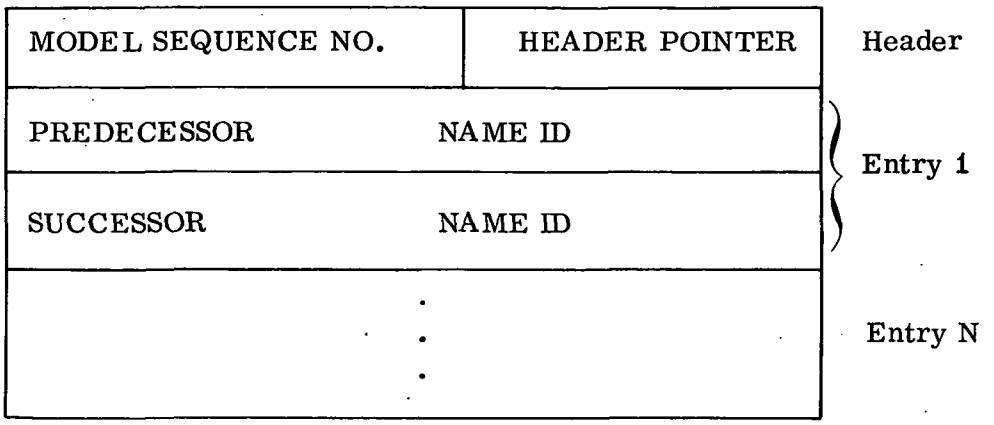
N_5 = Number of parameters.

N_6 = Number of connections.

2. Sequence of tables is the same as they are stored in the file.

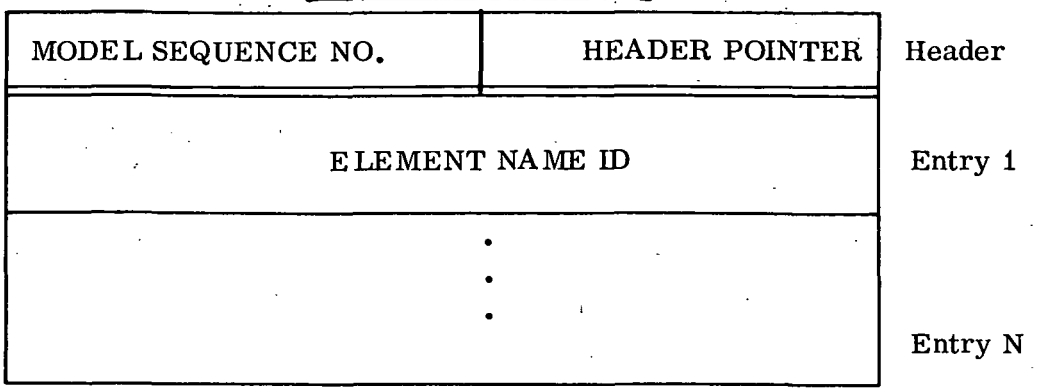
NAME	CONTENTS
T-NAME DICTIONARY	Cross-reference between each (long) MARSYAS name and (short) packed identification code word NAME ID.
T-MODEL INPUT TABLE	List of all input terminals of a model.
T-MODEL OUTPUT TABLE	List of all output terminals.
T-ELEMENTS TABLE	List of all elements of a model in short form (i.e., ELEMENT NAME ID).
T-PARAMETER TABLE	List of all parameters of all elements.
T-CONNECTION TABLE	List of all connections between two terminals.

T-CONNECTION TABLE



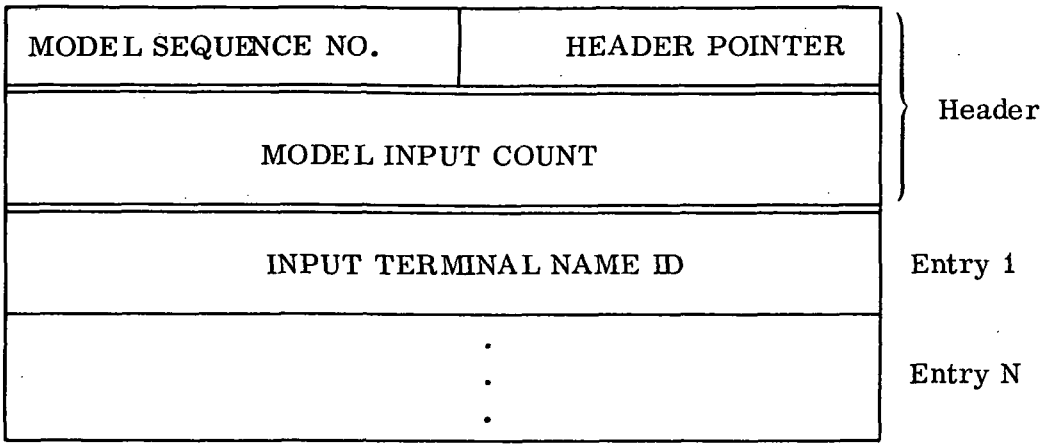
NAME	CONTENTS
MODEL SEQUENCE NO.	Sequence number of the model.
HEADER POINTER	Number of UNIVAC 1108 words in T-CONNECTION TABLE.
PREDECESSOR NAME ID	See structure overview of NAME ID.
-, SEQUENCE NO.	Sequence number assigned to model or element to which the terminal belongs.
-, PREFIX	"Undefined," "model," or "element."
-, SUFFIX	"Model input terminal" or "element output terminal."
-, SUBSCRIPT	"Any terminal number."
SUCCESOR NAME ID	
-, SEQUENCE NO.	Sequence number assigned to model or element to which the terminal belongs.
-, PREFIX	"Undefined" or "model" or "element."
-, SUFFIX	"Model output terminal" or "element input terminal"
-, SUBSCRIPT	"Any terminal number."

T-ELEMENTS TABLE



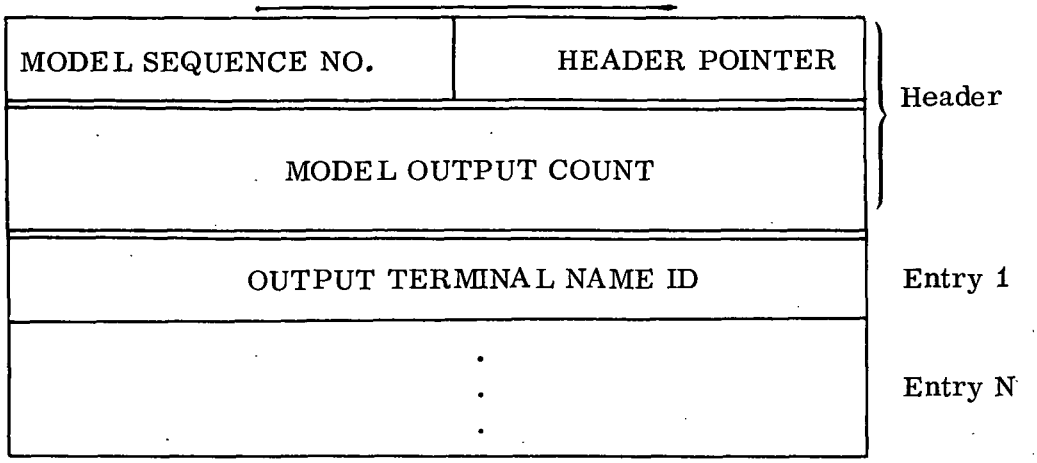
NAME	CONTENTS
MODEL SEQUENCE NO.	Sequence number of the model.
HEADER POINTER	Number of UNIVAC 1108 words in T-ELEMENTS TABLE.
ELEMENT NAME ID	Packed name identification code word (see structure overview of NAME ID).
-, SEQUENCE NO.	Sequence number assigned to element.
-, PREFIX	"Undefined," "defined," or "pointer."
-, SUFFIX	"Element."
-, SUBSCRIPT	"Any element type."

T-MODEL INPUT TABLE



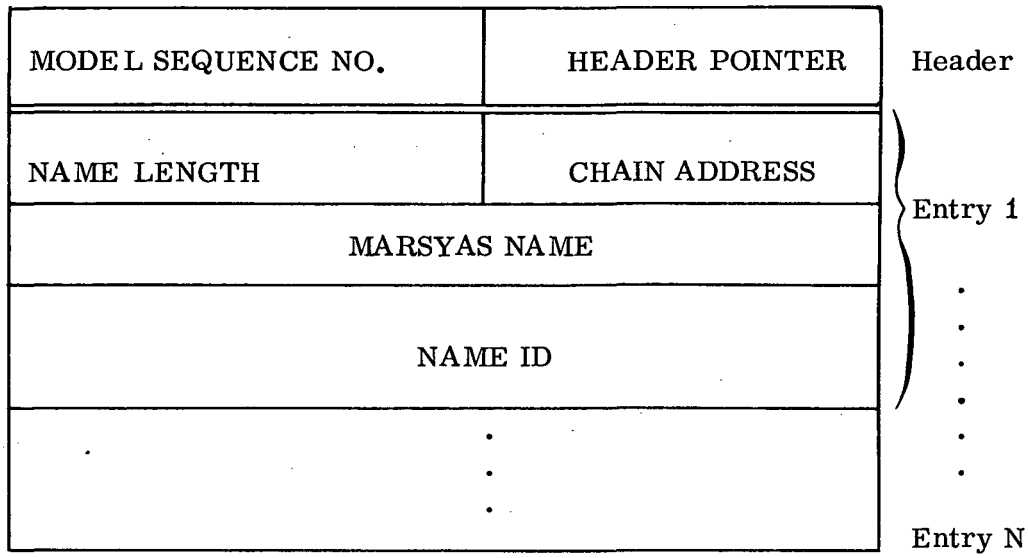
NAME	CONTENTS
MODEL SEQUENCE NO.	Sequence number of the model.
HEADER POINTER	Number of UNIVAC 1108 words in T-MODEL INPUT TABLE.
MODEL INPUT COUNT	Number of model input terminals.
INPUT TERMINAL NAME ID	See structure overview of NAME ID.
-, SEQUENCE NO.	Sequence number of model.
-, PREFIX	"Undefined," "defined," or "pointer."
-, SUFFIX	"Model input terminal."
-, SUBSCRIPT	"Any terminal number."

T-MODEL OUTPUT TABLE



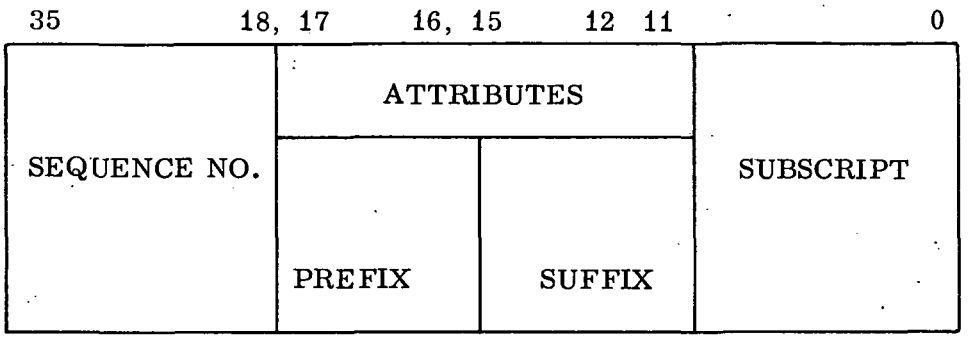
NAME	CONTENTS
MODEL SEQUENCE NO.	Sequence number of the model.
HEADER POINTER	Number of UNIVAC 1108 words in T-MODEL OUTPUT TABLE.
MODEL OUTPUT COUNT	Number of model output terminals.
OUTPUT TERMINAL NAME ID	See structure overview of NAME ID.
-, SEQUENCE NO.	Sequence number of model.
-, PREFIX	"Undefined," "defined," or "pointer."
-, SUFFIX	"Model output terminal."
-, SUBSCRIPT	"Any terminal number."

T-NAME DICTIONARY



NAME	CONTENTS
CHAIN ADDRESS	Address of next entry (begin).
HEADER POINTER	Number of UNIVAC 1108 words in T-NAME DICTIONARY plus 2.
MARSYAS NAME	MARSYAS - name of up to 36 characters.
MODEL SEQUENCE NO.	Sequence number of the model.
NAME ID	Packed name identification code word (see separate structure overview). The code of the SUFFIX and SUBSCRIPT depend on whether the name belongs to a model, element, terminal, or parameter.

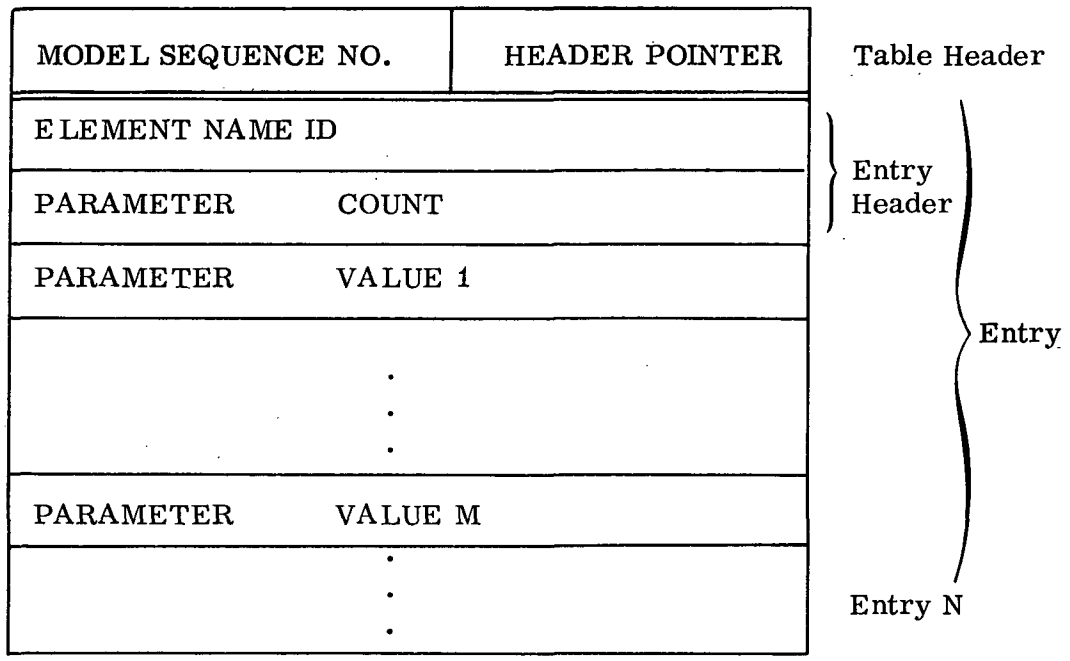
NAME ID[3]



NAME	CONDITION	CONTENTS
SEQUENCE NO.	-	Sequence number assigned to each model and each element.
PREFIX	-	"Undefined," "defined," or "pointer."
SUFFIX	-	"Undefined," "model," or "element." "Model input terminal," "model output terminal," "element input terminal," "element output terminal," or "element parameter."
SUBSCRIPT	(SUFFIX) = undefined	"0"
	(SUFFIX) = model	"0"
	(SUFFIX) = element	"Any element type" (e.g., "adder")
	(SUFFIX) = ...terminal	"Any terminal number."
	(SUFFIX) = element parameter	"Any parameter position."

3. The NAME-ID word is used in most MARSYAS tables; it is UNIVAC 1108 word-oriented.

T-PARAMETER TABLE



NAME	CONTENTS
MODEL SEQUENCE NO.	Sequence number of the model.
HEADER POINTER	Number of UNIVAC 1108 words in T-PARAMETER TABLE.
ELEMENT NAME ID	See structure overview of NAME ID.
-, SEQUENCE NO.	Sequence number of the element the parameters belong to.
-, PREFIX	"Undefined," "defined," or "pointer."
-, SUFFIX	"Element."
-, SUBSCRIPT	"Any element type."
PARAMETER COUNT	Number of parameters per element (=M)
PARAMETER VALUE i	Value of ith parameter in floating point or $11\ldots11_2$ if parameter is referenced by name.

3. Format of Tables and Comments

a. MODEL TABLES FILE

(1) Format. The tables of the MODEL TABLES FILE are stored for each model in the same sequence as depicted in the TSO. In the Functional Data Base (FDB), they are stored one file after the other as the models have been entered into the FDB.

All tables are of variable length. However, at the end of the DESCRIPTION PROGRAM MODULE and MODIFICATION PROGRAM MODULE, the tables of the MODEL TABLES FILE have been completed and their length is known.

All headers are of the same format; i.e., the first UNIVAC 1108 computer word of a table is the sequence number of 18 bits assigned to the model and the second computer word is the header pointer which indicates the length of the table. An entry is deleted from a table by simply overwriting it with binary 1's; i.e., giving a word the value 2^{36} -1.

(2) Comments. The MARSYAS processor represents a model internally by the six model tables given in the TSO.

The headers are used to step easily through the MODEL TABLES FILE to find the appropriate table.

The NAME ID is employed in constructing entries in all of the tables. The only tables which have entries containing information which is not in the form of a NAME ID are the PARAMETER TABLE (which, besides the ID, has numeric entries) and the NAME DICTIONARY (which has alphanumeric entries).

Each table is composed of a header and the table proper. The header consists of several computer words immediately preceding the table proper. It provides information such as the model number and the number of entries in the table. The PARAMETER TABLE has additional headers contained inside the table proper.

During the time the DESCRIPTION or MODIFICATION PROGRAM MODULE runs, the model tables are in the process of being constructed and each of the tables is open-ended. When the description of the model has been completed, its tables become closed. (Closing the tables requires no special action if the table headers have been properly maintained.) In the local dictionary approach, these tables are then moved adjacent to one another, thereby providing a description of the model in one unit. In the global dictionary approach, there is only one NAME DICTIONARY for all the models, and closing a model's dictionary actually means closing a section of the global DICTIONARY and keeping the various model tables distinct in their global tables.

b. T-CONNECTION TABLE

(1) Format. The header of the CONNECTION TABLE is of the same format as the headers of all other model tables; i.e., the first UNIVAC 1108 computer word contains the model sequence number, and the second word contains the header pointer.

An entry in the CONNECTION TABLE consists of two computer words. The first word of the entry is the NAME ID of the predecessor. This ID may be of an element or submodel output terminal, of a model input terminal, or of an element (if the element has one input and one output). The second word of the entry is the NAME ID of the successor. This ID may be of an element or submodel input terminal, of a model output terminal, or of an element (if the element has one input and one output). (See TSO.)

(2) Comments. The TEMPORARY CONNECTION TABLE and TEMPORARY DISCONNECT TABLE are tables used in core during the DESCRIPTION or MODIFICATION for generating the final CONNECTION TABLE of a model. They are not stored with the model, in the FDB or otherwise. They are described separately in the documentation.

c. T-ELEMENTS TABLE

(1) Format. The entries in the ELEMENTS TABLE are one UNIVAC 1108 computer word long, consisting of the element or submodel ID. (See TSO.)

(2) Comments. Although deletions may be made from this table (and from any other), no necessity exists for a model element count because that number is never used.

d. T-MODEL INPUT TABLE

(1) Format. The MODEL INPUT COUNT and the entries are one UNIVAC 1108 computer word long. (See TSO.)

(2) Comments. The MODEL INPUT COUNT word contains a count of the model input terminals. This value may not be computable from the header pointer if terminals have been deleted and, hence, is necessary.

e. T-MODEL OUTPUT TABLE

Format and comments are analogous to T-MODEL INPUT TABLE. (See TSO.)

f. T-NAME DICTIONARY

(1) Format. Since a MARSYAS name is of variable length, provision is made for efficient utilization of storage by grouping the NAME DICTIONARY into six subdictionaries according to the length of a name.

A MARSYAS name may be at most 36 characters long. In the UNIVAC 1108, a character is represented by six-bits, and a word is 36-bits long. Thus, a MARSYAS name may be up to six complete UNIVAC 1108 words long. Names which are at most one word long are placed in the first subdictionary, names which are more than one word and at most two words are placed in the second subdictionary, and so on, up to names which are more than five and at most six words long, which go into the sixth subdictionary. When the MARSYAS name does not consist of an integral number of words, the unused bits in the last words are set to zero. (See TSO.)

(2) Comments. The header pointer is used in two ways. During the DESCRIPTION and MODIFICATION MODULES, when the model is being described and the tables are open-ended, the header-pointer provides the location of the first word of the next entry in subdictionary. Thus, for example, if a 19 character name is to be entered in the dictionary, it is first filled out with zeros to occupy four computer words. Then it is entered into the four-word subdictionary with its NAME ID, beginning at the location obtained by adding the header-pointer to the address of the first header word. When an entry is made in a subdictionary, the header pointer should be increased by the length of the entry. Thus, after the above four-word name is entered in the subdictionary, the header-pointer is increased by six.

The second use of the header-pointer is for searching the NAME DICTIONARY. If, for example, it is desired to find the ID number of a name of 19 characters, the name is first filled out with zeros to occupy four computer words. Once the NAME DICTIONARY has been accessed, the header-pointer of the first subdictionary is used to access the header of the second subdictionary; the new header-pointer is used to access the header of the third subdictionary, and, similarly, the header of the fourth subdictionary is found. Then, the program steps through the entries of the fourth subdictionary, comparing each entry's alphanumeric section with the given word, until a match is found. The ID of the match entry is then easily accessed.

For this search procedure to work, it is necessary that the ordering of the subdictionaries be known. Thus, the two-word subdictionary should immediately follow the one-word subdictionary, and the three-word subdictionary should follow the two-word subdictionary, etc.

The search procedure described above may also be used to access a desired table. The model tables are placed in a fixed order, that of the list at the beginning of this section. Thus, to access the model ELEMENTS TABLE, the program goes to the beginning of the entry for the model in the FDB, accesses the one-word subdictionary header, steps through the subdictionaries, uses headers to step through the INPUT and OUTPUT TABLES, and arrives at the ELEMENTS TABLE header.

g. NAME ID

(1) Format and Comments. The bit length of each field is given in the TSO. The field formats are presented in this paragraph.

(2) SEQUENCE NO. The 18-bit sequence number field contains the sequence number of the object addressed by the NAME ID. Each model and element receive a unique sequence number. Objects such as input /output terminals and element parameters use the sequence number of their associated element or model.

The sequence number of a new element or model is assigned by the MARSYAS processor, using the value stored in the indicator I-SEQUENCE NUMBER. At the beginning of a user program, the I-SEQUENCE NUMBER is initialized at the first free value; i.e., at one more than the highest sequence number used in the FDB. Each time a new sequence number is assigned, the I-SEQUENCE NUMBER is incremented by one.

(3) PREFIX. The permissible prefix values and the conditions they stand for are as follows:

00 - 'Undefined.'

01 - 'Pointer.'

11 - 'Defined.'

A PREFIX set to 'defined' means that the MARSYAS name has been used in a context which generates all the necessary NAME ID information. For example, when an element name appears in an ELEMENT statement, it is assigned a 'defined' prefix. Since, at the end of the DESCRIPTION and MODIFICATION MODULES, all names should be properly defined, the PREFIX field, as used here, becomes extraneous. It may be decided to store other information in this field for other modules. (If necessary, the ID code may be redesigned for this purpose to allow three bits in the prefix field and three bits in the suffix field.)

A PREFIX set to 'undefined' means that the MARSYAS name has been used, but only in contents which do not define it completely. This condition arises even in valid MARSYAS programs, although temporarily. This is a result of the source program statements that are processed sequentially by the MARSYAS processor, and the user may choose to use a name in a

statement appearing before the statement in which it is completely defined. For example, in the instructions,

ELEMENTS, AD, ALPHA.

CONNECT, ALPHA, BETA.

INPUTS, BETA.

The MARSYAS name BETA will be 'undefined' after the connect statement is read. The prefix will change to defined when the next statement is read. The element ALPHA will have a 'defined' prefix after the first statement.

If an 'undefined' prefix is used, the sequence number field is zero. A PREFIX set to 'pointer' means that the MARSYAS name has been defined in terms of another name (by a NAMING statement), and as yet neither of these names has been defined completely. Also, this is a temporary prefix. An example of situations in which it arises is the following:

NAMING, GAMMA, DELTA.

ELEMENTS, AD, DELTA.

GAMMA is designated by the user to represent the same object as DELTA, in the user NAMING statement. In the above example, DELTA is still undefined when the NAMING statement is processed. Hence, the ID prefix for GAMMA is set to 'pointer.' The sequence number field of GAMMA ID is set to the address of the ID of DELTA in the NAME DICTIONARY. In general, this is the meaning of the SEQUENCE NUMBER for a pointer prefix; i.e., the sequence number field contains the address of the ID which defines the original name. When DELTA has been defined in the succeeding ELEMENTS statement, the GAMMA ID can be changed to a defined ID.

(4) SUFFIX. The SUFFIX field of the ATTRIBUTES' FIELD supplies the general description of the object type. The possible codes and their meanings are as follows:

0000 - 'Undefined' (temporary designation).

1000 - 'Model.'

1001 - 'Model input terminal.'

1010 - 'Model output terminal.'

1100 - 'Element.'

1101 - 'Element input terminal.'

1110 - 'Element output terminal.'

0100 - 'Element parameter.'

When used with a prefix of 'defined,' these suffix codes describe the defined object type. When the prefix is 'undefined' or 'pointer,' these codes refer to attributes which have been implied by the context of usage. For example, if an undefined name EPSILON appears in the statement:

PARAMETER, BLOCK, EPSILON, 01.

The suffix "element parameter" is implied in the EPSILON ID.

For the subscript field, the distinction between defined and implied characteristics is the same.

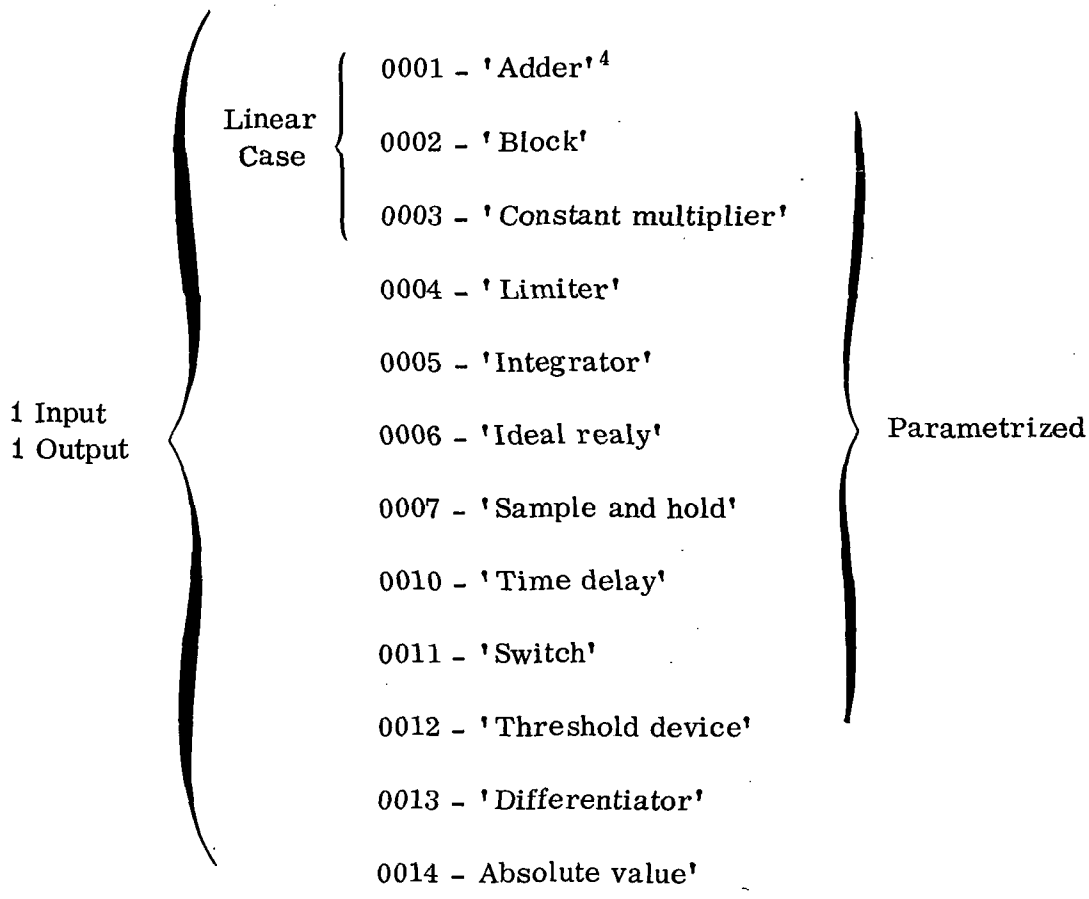
(5) SUBSCRIPT. The meaning of the SUBSCRIPT depends on the attributes field SUFFIX.

If SUFFIX is 'undefined,' the subscript field is zero. The whole name ID should be zero.

If SUFFIX is 'model,' the subscript field is zero. (Exception: See note in description of SUBMODEL operator.)

If SUFFIX is 'element,' the subscript field contains a code for the element class. The code is as follows (numbers are in octal.):

0000 — 'Undefined'



If SUFFIX is a terminal (model or element, input or output), the subscript field is the terminal number. Thus, in the present ID code, a model may have a maximum of 2^{12} -1 input or output terminals.

4. The input terminals of an adder are not distinguished.

If SUFFIX is 'element parameter,' the subscript field is the parameter number (i.e., its numerical position in the element parameter list).

h. T-PARAMETER TABLE

Format and Comments. An entry is made in the table for each parameter element and its parameters. Since entries are of variable length, each entry is given a header (element header). The element header is two words. The first word contains the element NAME ID. The second word contains the number of parameters required by the element. This, in effect, is a pointer to the end of the entry. (See TSO.)

The values of the parameters of the element immediately follow the element header. Each numeric value is stored in single precision floating-point format. The only exception is the first parameter of a block, which is stored in fixed point format.

During the DESCRIPTION or MODIFICATION MODULES, a parameter value may be set to 2^{36} -1 ($11....11_2$). This is not the true numeric value, but a code. The code indicates that as yet no numeric value has been given to the parameter. This condition is only temporary, and in defined models, such as models stored in the FDB, does not occur.

The TEMPORARY PARAMETER TABLE is not to be confused with the PARAMETER TABLE. The former is a table used to generate the final status of the PARAMETER TABLE during the DESCRIPTION or MODIFICATION of a model.

APPENDIX B. AN EXAMPLE OF AN "INDICATOR LIST"

Name (FORTRAN Name)	Function - Contents - Format
I-ALPHA: (IALPM)	Stores 'any NAME ID' of the element given at the beginning of the argument of a PARAMETER statement; one UNIVAC 1108 word, binary, format of NAME ID.
I-ATTRIBUTES SWITCH: (IATTSW)	Stores 'any NAME ID' to be assigned to an element name; one UNIVAC 1108 word, binary, format of NAME ID.
I-BASIC ID: (IBASID)	Stores 'any NAME ID' to be assigned to parameters of an ELEMENTS statement or to model tables in an INPUTS or OUTPUTS statements; one UNIVAC 1108 word, binary, format of NAME ID.
I-BTABLE: (IBTABL)	Stores 'any address' of the first location of a table which is being referenced; one UNIVAC 1108 word, binary code.
I-CHARACTER COUNT: (ICNT)	Counts the 'length of a user word' excluding word terminator; one UNIVAC 1108 word, binary.
I-NUMBER: (INO)	Contains designation of internal representation of a number; i.e., ' number type' : 1_{10} = floating point, 2_{10} = fixed point, 3_{10} = integer, 0 = unknown; one UNIVAC 1108 word.

APPENDIX C. Extract of "Software Programming Specifications" from DESCRIPTION Program Module of MARSYAS

Description of the subprograms S-PLACE WORD, S-SCAN CHARACTER, and part of S-SCAN WORD (function groups P'1, P'2, and P'3) is given in this appendix.

1. Processing Description

S-PLACE WORD

* C1.1	:	(I-OPERATOR INDICATOR) = 'element mnemonic'
	P1 :	1. (I-POINTER MODIFIER) ← '5'
		2. GO TO P2, 2
* C1.2	:	(I-OPERATOR INDICATOR) ≠ 'element mnemonic'
	P2 :	1. (I-POINTER MODIFIER) ← '0'
		2. (I-WORK AREA POINTER) ← ' begin OF B-WORK AREA'
		3. GO TO P3,4
* C2.1	:	(I-INPUT AREA POINTER) + (I-POINTER MODIFIER) + (I-CHARACTER COUNT)
		+ (I-RIGHT PAREN) > '73'
	P3 :	1. write (B-INPUT AREA) into printer
		2. (B-INPUT AREA) ← '0'
		3. (I-INPUT AREA POINTER) ← '15'
		4. GO TO P4,1
* C2.2	:	(*C2.1) ≤ '73'
	P4 :	1. (B-INPUT AREA, (I-INPUT AREA POINTER) + (I-POINTER MODIFIER)) ← 0
		(B-WORK AREA, (I-WORK AREA POINTER))
		0 until (I-INPUT AREA POINTER) = '(CHARACTER COUNT) + (RIGHT PAREN) + 1'
		2. (I-INPUT AREA POINTER) ← (I-INPUT AREA POINTER) +
		(RIGHT PAREN) + (CHARACTER COUNT) + 2
←	E1 :	return

S-SCAN CHARACTER

* C1.1	:	(I-SCPOINTER) > 'any input record buffer length'
* C2.1	:	(H-INPUT RECORD) ≠ 'available'
		(I-ERROR INDICATOR' 8) ⟵ 'on'
⟵	R1 :	exit to S-ERROR ROUTINE
* C2.2	:	(C1.1) = 'available'
	P1 :	1. read next input record
		2. (I-SCPOINTER) ⟵ 'first character of record'
		3. GO TO P3,1
* C1.2	:	(* C1) ≤ 'input record buffer length'
	P2 :	1. (B-WORK AREA,(I-WORK AREA POINTER)) $\xleftarrow[0]{}$ (B'-USER DECK, (I-SCPOINTER)),
		0 skipping blanks.
		2. (I-SCPOINTER) ⟵ (I-SCPOINTER) + '1'
⟵	E1 :	3. return

S-SCAN WORD (P'1, P'2, P'3)

		P1 :	1. (B-WORK AREA) ⟵ '0'
			2. (I-WORK AREA POINTER) ⟵ '0'
			3. (I-CHARACTER COUNT) ⟵ '-1'
(S1)		P2 :	1. call S-SCAN CHARACTER
			2. (I-CHARACTER COUNT) ⟵ (I-CHARACTER COUNT) + '1'
	* C1.1	:	(B-WORK AREA, (I-CHARACTER COUNT)) ≠ 'any TERMINATOR' or '=' or '*'
		P3 :	(I-WORK AREA POINTER) ⟵ (I-WORK AREA POINTER) + '1'
	* C2.1	:	(I-WORK AREA POINTER) ≤ '36'
			GO TO P2,1
	* C2.2	:	(* C2.1) > '36'
			(I-ERROR INDICATOR 16) ⟵ 'on'
	⟵	R1 :	exit to S-ERROR ROUTINE

S–SCAN WORD

* C1.2	:	(B–WORK AREA, (I–CHARACTER COUNT)) = 'any terminator'
	P4 :	(I–TERMINATOR SWITCH) ⟵ 'any terminator'
* C3.1	:	(I–TERMINATOR SWITCH) = '('
	P5 :	1. (I–PAREN COUNT) ⟵ (I–PAREN COUNT) + '1'
		2. GO TO * C4
* C3.2	:	(* C3.1) ≠ '('
* C4.1	:	(B–WORK AREA, (I–CHARACTER COUNT) – 1) = ')'
	P6 :	1. (I–PAREN COUNT) ⟵ (I–PAREN COUNT) – '1'
		2. (I–CHARACTER COUNT) ⟵ (I–CHARACTER COUNT) – '1'
		3. (I–RIGHT PAREN) ⟵ '1'
		4. GO TO * C5
* C4.2	:	(* C4.1) ≠ ')'
	P7 :	(I–RIGHT PAREN) ⟵ '0'
* C5.1	:	(B–WORK AREA, CHARACTER 1) ≠ 'any numeric' or 'any alphabetic'
		(I–ERROR INDICATOR 16) ⟵ 'on'
⟵	R2 :	exit to S–ERROR SUBROUTINE

S-SCAN WORD

	* C5.2	:	(* C5.1) = 'any numeric'
	* C6.1	:	(I-EXPECTED WORD TYPE) ≠ 'numeric'
			(I-ERROR INDICATOR 16) ← 'on'
	←	R3 :	exit to S-ERROR SUBROUTINE
	* C6.2	:	(* C6.1) = 'numeric'
	* C7.1	:	(I-TERMINATOR SWITCH) = '.'
(S1)		P8 :	call S-SCAN CHARACTER
	* C8.1	:	(B-WORK AREA, (I-CHARACTER COUNT)) = 'any numeric'
			GO TO P2,2
	* C8.2	:	(* C8.1) ≠ 'any numeric'
		P9 :	(I-SCPOINTER) ← (I-SCPOINTER) - '1'
			GO TO P10,1
	* C7.2	:	(I-TERMINATOR SWITCH) ≠ '.'
(S2)		P10 :	1. call S-PLACE WORD
(S3)			(2. call S-CALCULATE VALUE)
	←	E1 :	3. return
	* C5.3	:	(B-WORK AREA, CHARACTER1) = 'any alphabetic'
	←	E2 :	exit to NEXT ROUTINE
	* C1.3	:	(B-WORK AREA, (I-CHARACTER COUNT)) = '=' or '*'
	←	E3 :	exit to NEXT ROUTINE

2. Comments

a. S-PLACE WORD

P1 - The POINTER MODIFIER is set to 5 since the argument list for an element begins in column 20 rather than 15.

P3 - If the last character of the argument word were to go beyond column 73, the B-INPUT AREA (without the present argument) is printed and then set to all blanks. In the listing, columns 15 through 73 are reserved for the argument list. The arguments are printed one after another on the same line, until this area is about to overflow, when a new line is begun.

P4 - Transfer the characters of the argument from B-WORK AREA to B-INPUT AREA.

b. S-SCAN CHARACTER

P2 - The nonblank character is placed into the appropriate position in B-WORK AREA by using I-WORK AREA POINTER.

c. S-SCAN WORD

(1) General. DEFINITION OF (WORD): For the purposes of S-SCAN WORD, 'word' is used in a special sense a word is a set of alphanumeric characters which is read in from the user code and processed in one execution of S-SCAN WORD. As such, a word begins with the alphanumeric character directly following the last character process; it ends with a terminator. The S-SCAN WORD terminators are ".", ",", and "(." For example, in the MARSYAS statement;

INPUTS, ALPHA, BETA,

the words are 'INPUTS,' 'ALPHA,' and 'BETA.' Also, the expression

ELEMENT (U10)

gives rise to the two words 'ELEMENT' and 'U10).'

S-SCAN WORD assumes that the I-EXPECTED WORD TYPE indicator has been set to show which word types are permissible in the current context. In most contexts, only one word type will be permissible. For example, after a period, an operator must follow so the I-EXPECTED WORD TYPES is set to 'OPERATOR.' However, in some situations, more than one possibility exists. For instance, in the parameter list of an ELEMENTS statement, a parameter may be designated by a number or by the name of the parameter. In this case, I-EXPECTED WORD TYPE will be set to 'name' or 'number.'

GENERAL LOGIC: S-SCAN WORD reads the user word, character by character, placing it in the B-WORK AREA. The reading terminates when a word terminator ['.', ',', or '('] is read, or, in the special cases of a COMMENT or = operator, when a '*' or '=' is read. The processing then branches on the first character of the word. If the first character is numeric, the word is processed as a number and its value is calculated. If the first character is alphabetic, the word is processed as an operator, mnemonic, standard function, subscript, or name, depending on the I-EXPECTED WORD TYPE. For the first three word types, the basic output of S-SCAN WORD is contained in the I-OPERATOR INDICATOR. This is set to the location of the operator, mnemonic, or standard function routine. For a name, the address of the name in the T-NAME DICTIONARY and its NAME ID are returned. For a subscript, the subscripted name ID is adjusted to its appropriate reading.

(2) Detail:

P1 — I-CHARACTER COUNT is initialized to -1, so that the terminator of the word will not be included in the count.

P2, P3 — Using S-SCAN CHARACTER, the next nonblank character is read into B-WORK AREA until a 'terminator' (i.e., ".", ",", "(", "*", or "=" is encountered, which means that a complete word is being read. In this process, the I-WORK AREA POINTER is tested to be less than or equal to 36. If it is not, the word has exceeded the MARSYAS word-length limit, and an error condition is generated (R1).

P4 — After one complete word has been read, the terminator symbol is stored in I-TERMINATOR SWITCH for subsequent processing by the subprogram which called S-SCAN WORD.

P5 — If the terminator is "(", the I-PAREN COUNT is increased by 1.

P6 — If the character preceding the terminator is '(,' the I-PAREN COUNT is decreased by 1. (This is the only valid position for a right parenthesis.) Also, the I-CHARACTER COUNT is decreased by 1 so that it addresses the word proper in B-WORK AREA. Further, the I-RIGHT PAREN indicator is set to 1. (This counter is used, in conjunction with the I-CHARACTER COUNT, to determine the total number of characters in the B-WORK AREA, when they are to be moved to the B-INPUT AREA for printing.)

P7 — If the character preceding the terminator is not a '),' I-RIGHT PAREN is set to zero.

*C5 — C5 checks the first character of the scanned word. C5.2 states that the first character is numeric (+, -, 1, 2, . . ., 9, 0).

*C7, P8 C7 checks if the terminator is ".". However, the "." can also be a decimal point. By examining the character following the ".", one can find out if the "." is a terminator or decimal point. Therefore, S-SCAN CHARACTER is called to read the next character.

If the character is a digit, the period actually was a decimal point. S-SCAN WORD goes back to the reading of the user word in order to complete the number (P2).

P9 If the character is not a digit, I-SCPOINTER is decreased by 1. (I-SCPOINTER is the pointer used by the S-SCAN CHARACTER to access the next character. It must be reset so that the next word to be read by S-SCAN WORD begins with the proper character.)

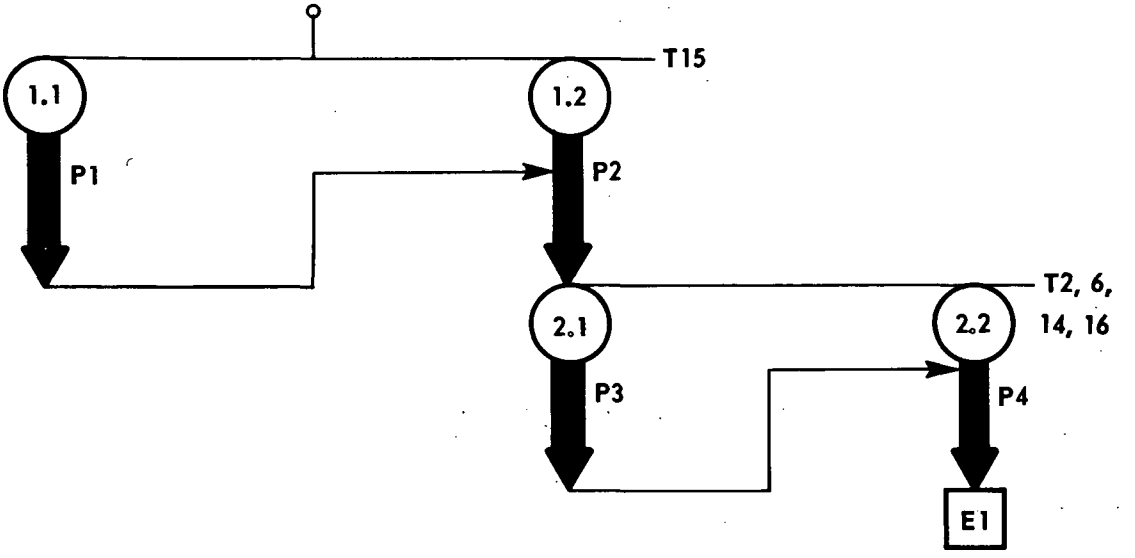
P10 When the numeric argument has been completed (or it was complete to begin with), the following processing occurs:

S-PLACE WORD is called. This routine places the alphanumeric word, with its terminator, in the appropriate section of the B-INPUT AREA so it will appear in the listing of the user deck.

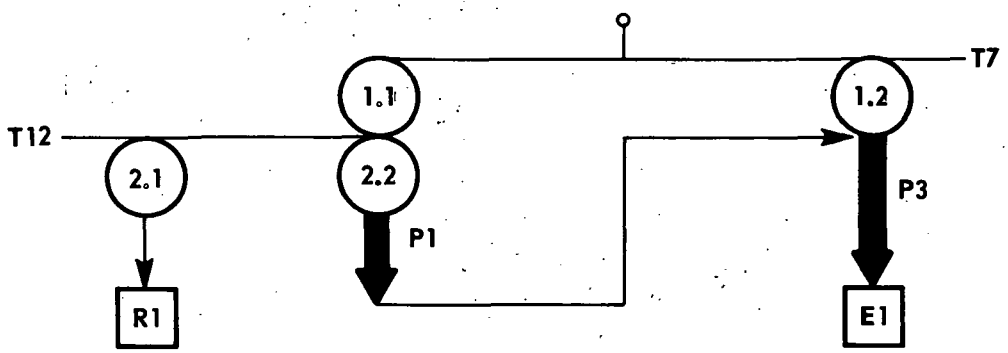
S-CALCULATE VALUE is called. This routine constructs the internal representation of the number. It sets the I-TYPE SWITCH to 'fixed point' or 'floating points,' depending on the type of the number. It also returns the internal machine representation of the number.

3. Flow Graphs A

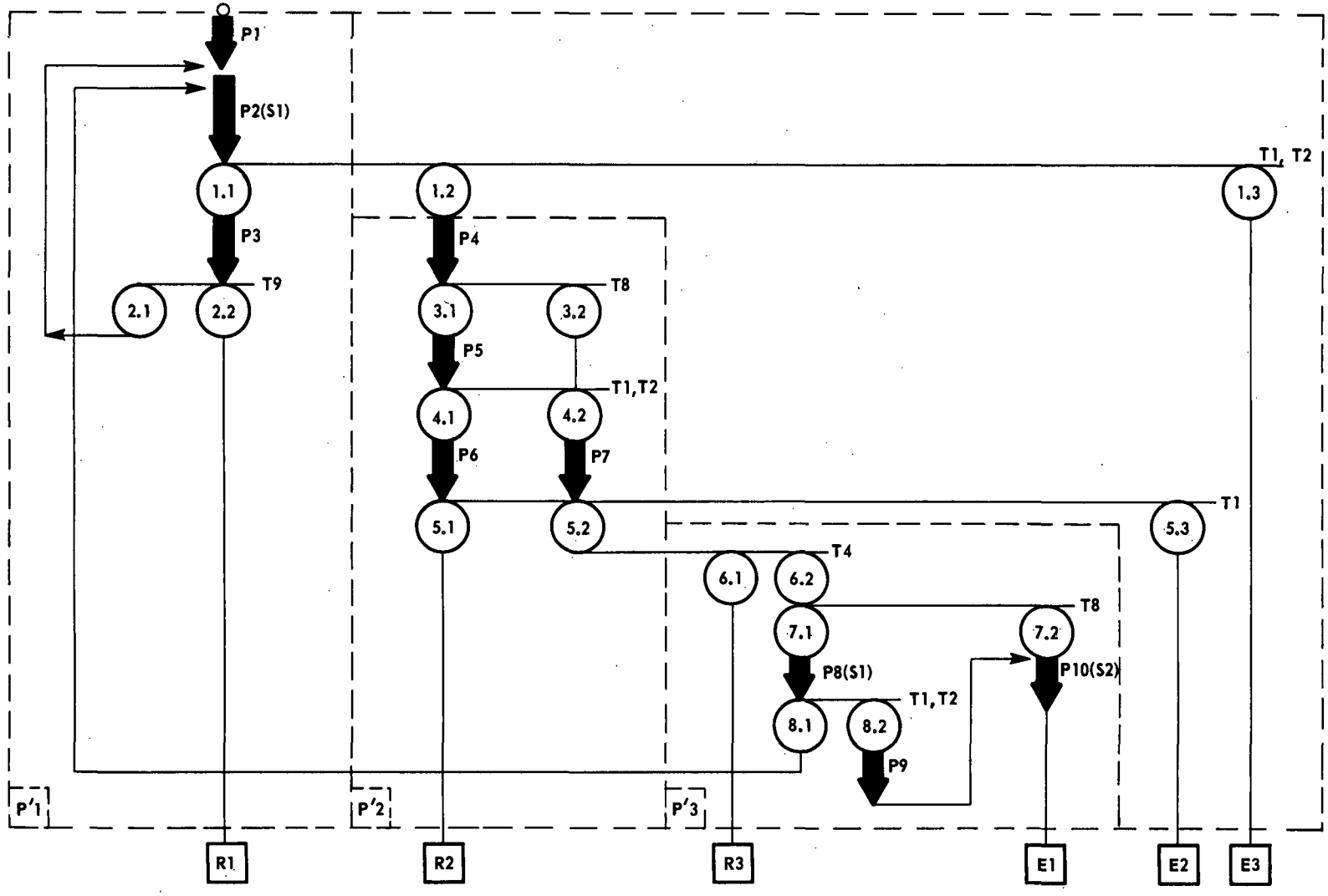
S-PLACE WORD



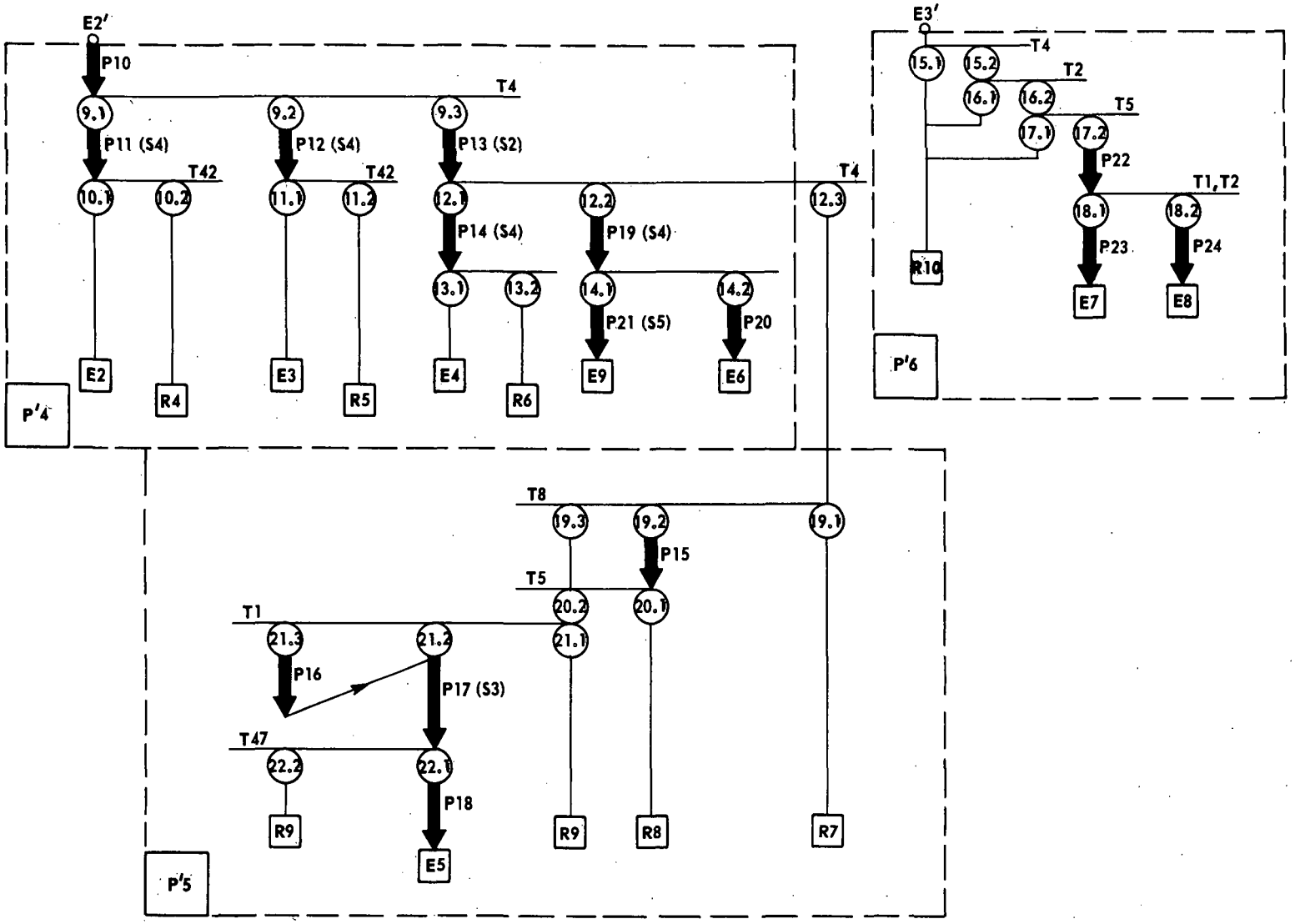
S-SCAN CHARACTER



S-SCAN WORD (P'1, P'2, P'3)



S–SCAN WORD (P'4, P'5, P'6)



4. Processing/Table Affect Diagram (PTAD)

S–PLACE WORD (PTAD)

	1	2	3	4	5	6	7	8
	B–INPUT AREA	B–WORK AREA	I–CHARACTER COUNT	I–INPUT AREA POINTER	I–OPERATOR INDICATOR	I–POINTER MODIFIER	I–RIGHT PAREN	I–WORK AREA POINTER
	T13	1	2	14	15	16	6	9
P1						1		
P2						1		1
P3	1			1				
P4	1	0	0	0 1		0	0	0
INPUT TABLE SET I		0	0	0	0		0	
OUTPUT TABLE SET 0 (E1)	1			1		1		1

(← ALPHABETIC ORDER)

S–SCAN CHARACTER (PTAD)

	1 T B–USER DECK T 10	2 B–WORK AREA 1	3 I–ERROR INDICATOR 18 11	4 I–SCPOINTER 7	5 I–WORK AREA POINTER 9	6 H–INPUT RECORD 12	
P1				1			
P2	0	1		01	0		
INPUT TABLE SET	0			0	0	0	
OUTPUT TABLE SET							
0 (E1)		1		1			
0 (R1)			1				

S-SCAN WORD (PTAD)

	T1 B-WORK AREA	2 I-CHARACTER COUNT	3 I-ERROR INDICATOR 16	4 I-EXPECTED WORD TYPE	5 I-PAREN COUNT	6 I-RIGHT PAREN	7 I-SCPOINTER	8 I-TERMINATOR SWITCH	9 I-WORK AREA POINTER	S1 S-SCAN CHARACTER: 10 B-USER DECK	11 I-ERROR INDICATOR 18	12 H-INPUT RECORD	S2 S-PLACE WORD: 13 B-INPUT AREA	14 I-INPUT AREA POINTER	15 I-OPERATOR INDICATOR	16 I-POINTER MODIFIER
P1	1	1							1							
P2	1	01					01		0	0	1	0				
P3									01							
P4								1								
P5					01											
P6		01			01	1										
P7						1										
P8	1						01		0	0	1	0				
P9							01									
P10	0	0				0			1				1	01	0	1
INPUT TABLE SET				0	0		0			0		0		0	0	
OUTPUT TABLE SET																
0 (E1)	1	1			1	1	1	1	1		1		1	1		1
0 (E2), 0 (E3)	1	1			1	1	1	1	1		1					
0 (R1)	1	1	1		1	1	1	1	1		1					
0 (R2), 0 (R3)	1	1	1		1	1	1	1	1		1					
MERGED OUTPUT TABLE SET	1	1	1		1	1	1	1	1		1		1	1		1

APPENDIX D. Extract of "Subprogram Functional Specifications" from DESCRIPTION Program Module of MARSYAS

Functional description of subprograms S-PLACE WORD, S-SCAN CHARACTER, and part of S-SCAN WORD (P'1, P'2, and P'3) is given in this appendix.

1. Description of Functions

S-PLACE WORD

P1, P2 : Initialize pointers for locating word in B-INPUT AREA and B-WORK AREA.

P3 : If an argument word is larger than B-INPUT AREA, i.e., if it goes beyond column 73, then the contents of B-INPUT AREA is printed. The B-INPUT AREA and its pointer are reset.

P4 : Transfer the characters of the argument from B-WORK AREA to B-INPUT AREA.

S-SCAN CHARACTER

P1: Check for error condition.

P2, P3: Read and transfer one nonblank character from B-USER DECK to B-WORK AREA.

S-SCAN WORD (P'1, P'2, AND P'3)

P'1 : Initialize indicators and read characters from user's deck one by one into B-WORK AREA until a terminator (*C1.2) or '=' or '*' (C1.3) is read.

P'2 : Control the proper number of parentheses; i.e., '(' and ').'

P'3 : The word is treated as a number, since the first character of a scanned word is a 'numeric' (*C5.2). The word is converted from an integer, fixed point, or floating point decimal number into its proper UNIVAC 1108 binary format. The number is also printed in the listing at the appropriate place (E1).

S-SCAN WORD (P'4, P'5, P' 6)

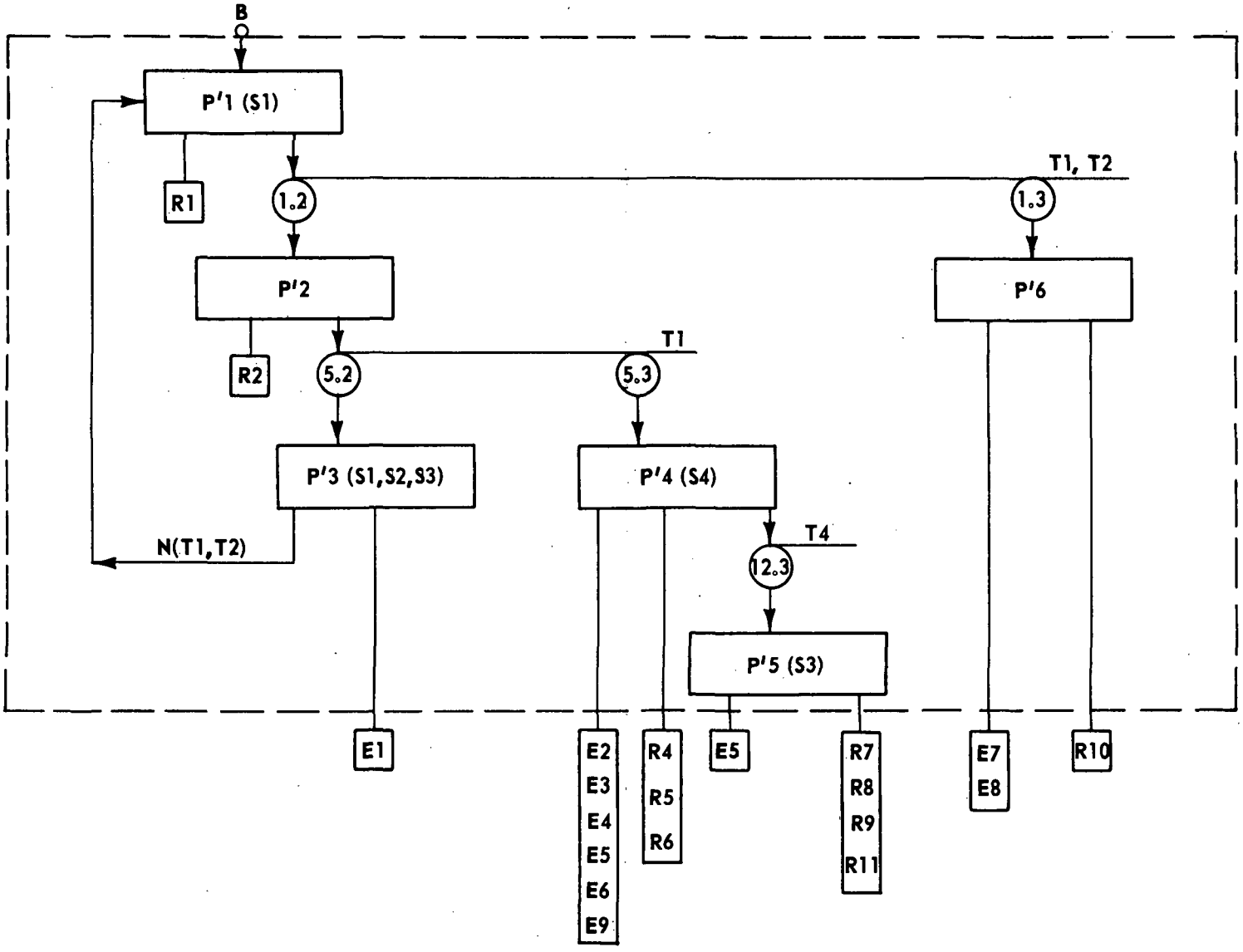
P'4 : The word is treated as an 'operator,' 'mnemonic,' or 'standard function' (*C5.3 : In either case, the word is printed in the listing at the appropriate place. The location of the subroutine to which control is being transfered is found in T-OPERATOR LIST - (E2), T-MNEMONIC LIST (E3), or T-STANDARD FUNCTION LIST (E4), respectively.

The word is treated as a 'name' (*C12. 2): In this case, T-NAME DICTIONARY is searched. If the same name is formed in this table, I-WORD TYPE SWITCH memorizes this fact (E6). Otherwise, the word is entered into T-NAME DICTIONARY as a new name (E9).

P'5 : The word is treated as a 'subscript' (*C12.3 : In this case, the suffix of the ID of the current entry of T-NAME DICTIONARY is filled with element, subscript ID, element output terminal, or element input terminal (E5).

P'6 : If the character is '=' or '*' (*C1.3 , the proper control switches are set to direct the control flow to S-CONTINUATION (E8) or S-COMMENT (E7), respectively.

S–SCAN WORD


B
P'1 (S1)
R1
1.2
T1, T2
1.3
P'2
P'6
R2
5.2
T1
5.3
P'3 (S1,S2,S3)
P'4 (S4)
N(T1,T2)
T4
12.3
P'5 (S3)
E1
E2
E3
E4
E5
E6
E9
R4
R5
R6
E5
R7
R8
R9
R11
E7
E8
R10

3. Input/Output Table Sets

S-PLACE WORD

Input table set = I [(T1) (T2) (T14) (T15) (T6)]

Valid Input Table States

(T1) = (B-WORK AREA) = 'Any data.'

(T2) = (I-CHARACTER COUNT) = 'Any number of characters.'

(T14) = (I-INPUT AREA POINTER) = 'Any number between 0 and 80.'

(T15) = (I-OPERATOR INDICATOR) = 'Element mnenomic,' 'no element mnenomic.'

(T6) = (I-RIGHT PAREN) = 'No right parenthesis,' 'right parenthesis.'

Resulting Output Table Set

O(E1) = O [(T13) (T14) (T16) (T9)]

S-PLACE WORD

Typical Input Table State Set	Resulting Typical Output Table State Set
$I\,[(T1)_1\,(T2)_1\,(T14)_1\,(T15)_1\;(T6)_1$ or $(T6)_2]$ (*)	$O\,(E1)_1$
$I\,[(T1)_1\,(T2)_1\,(T14)_1\,(T15)_2\;(T6)_1$ or $(T6)_2]$ (*)	$O\,(E1)_2$
$I\,[(T1)_1\,(T2)_2\,(T14)_2\,(T15)_2\;(T6)_1]$ (**)	$O\,(E1)_3$

(*) The number in T2 and T14 are chosen in such a way that they fulfill condition C 2.1.

(**) The numbers in T2 and T14 fulfill condition C 2.2.

S-PLACE WORD

Typical Output Table States

$O(E1)_1$: (T13) = (B-INPUT AREA) = 'Any data 1 of B-WORK AREA.'

(T14) = (I-INPUT AREA POINTER) = 'Any number calculated in P 4, 2.'

(T16) = (I-POINTER MODIFIER) = '5.'

(T9) = (I-WORK AREA POINTER) = 'Begin of B-WORK AREA.'

$O(E1)_2$: (T13) = (I-WORK AREA POINTER) = 'Any data 1 of B-WORK AREA.'

(T14) = (I-WORK AREA POINTER) = 'Any number calculated in P 4, 2.'

(T16) = (I-WORK AREA POINTER) = '0.'

(T9) = (I-WORK AREA POINTER) = 'Begin of B-WORK AREA.'

$O(E1)_3$: (T13) = (I-WORK AREA POINTER) = 'Any data 2 of B-WORK AREA.'

(T14) = (I-WORK AREA POINTER) = 'Any number calculated in P 4, 2.'

(T16) = (I-WORK AREA POINTER) = '0.'

(T9) = (I-WORK AREA POINTER) = 'Begin of B-WORK AREA.'

S-SCAN CHARACTER

Input table set = I [(T1) (T7) (T9) (T12)]

Valid Input Table States

(T1) = (B-USER DECK) = 'Any data.'

(T7) = (I-SCPOINTER = 'Any character position smaller than input record buffer length.'
'Any character position greater than input record buffer length.'

(T9) = (I-WORK AREA POINTER) = 'Any character position.'

(T12) = (H-INPUT RECORD) = 'Available,' 'not available' = $(T12)_1$, $(T12)_2$.

Resulting Output Table Sets

O (E1) = O [(T1) (T7)]

O (R1) = O [(T11)]

S-SCAN CHARACTER

Typical Input Table State Set	Resulting Typical Output Table State Set
I [$(T1)_1$ $(T7)_1$ $(T9)_1$ $(T12)_1$ or $(T12)_2$]	O $(E1)_1$
I [$(T1)_1$ $(T7)_2$ $(T9)_1$ $(T12)_1$]	O $(E1)_2$
I [$(T1)_1$ $(T7)_2$ $(T9)_1$ $(T12)_2$]	O $(R1)_1$

O $(E1)_1$: (T1) = (B-WORK AREA) = 'Any data of I (T2) without blanks.'

(T7) = (I-SCPOINTER) = 'Any character position of I (T7) + 1.'

O $(E1)_2$: (T1) = (I-SCPOINTER) = 'Any data of I (T2) without blanks.'

(T7) = (I-SCPOINTER) = '2. character position.'

O $(R1)_1$: (T11) = (I-ERROR INDICATOR 18) = 'On.'

S-SCAN WORD (P'1, P'2, P'3)

Input table set = I [(T4) (T5) (T7) (T10) (T12) (T14) (T15)]

Valid Input Table States

(T4) = (I-EXPECTED WORD TYPE) = 'Numeric,' 'no numeric' $(T4)_1$, $(T4)_2$ ('no numeric' = 'operator,' 'mnemonic,' 'subscript,' 'name,' 'standard function').

(T5) = (I-PAREN COUNT) = 'Any number of uncancelled left parentheses.'

(T7) = (I-SCPOINTER) = 'Any character position ≤ input record buffer length.'
'Any character position > input record buffer length.'

(T10) = (B-USER DECK) = 'Any MARSYAS word.'

(T12) = (H-INPUT RECORD) = 'Available,' 'not available' = $(T12)_1$, $(T12)_2$.

(T14) = (I-INPUT AREA POINTER) = 'Any number between 0 and 80.'

(T15) = (T-OPERATOR INDICATOR) = 'Element mnemonic,' 'no element mnemonic.'

Resulting Output Table States

O (E1) = O [(T1) (T2) (T5) (T6) (T7) (T8) (T9) (T11) (T13) (T14) (T16)] .

O (E2) = O (E3) = O [(T1) (T2) (T5) (T6) (T7) (T8) (T9) (T11)].

O (R1) = O (R2) = O (R3) = O [(T1) (T2) (T3) (T5) (T6) (T7) (T8) (T9) (T11)].

S-SCAN WORD

Typical Input Table State Set	Resulting Output Table State Set
$I[(T4)_1 (T5)_1 (T7)_1 (T10)_1 (T12)_1 (T14)_1 (T15)_1]$	$O(E1)_1$
$I[(T4)_2 (T5)_1 (T7)_1 (T10)_1 (T12)_1 (T14)_1 (T15)_1]$	$O(E1)_2$
$I[(T4)_1 (T5)_1 (T7)_2 (T10)_1 (T12)_1 (T14)_1 (T15)_1]$	$O(E1)_3$
$I[(T4)_1 (T5)_1 (T7)_1 (T10)_1 (T12)_2 (T14)_1 (T15)_1]$	$O(E1)_4$
$I[(T4)_1 (T5)_1 (T7)_1 (T10)_1 (T12)_1 (T14)_1 (T15)_2]$	$O(E1)_5$

S-SCAN WORD

Typical Output Table States

$O\ (E1)_1$: (T1) = (B-WORK AREA) = 'Any data of I (T10) without blanks.'

(T2) = (I-CHARACTER COUNT) = 'Length of the scanned word in B-WORK AREA without terminators.'

(T5) = (I-PAREN COUNT) = 'Number of uncancelled parentheses.'

(T6) = (I-RIGHT PAREN) = '1' or '0.'

(T7) = (I-SCPOINTER) = 'Any character position in word.'

(T8) = (I-TERMINATOR SWITCH) = 'Any terminator.'

(T9) = (I-WORK AREA POINTER) = 'Begin of B-WORK AREA.'

(T11) = (I-ERROR INDICATOR 18) = 'Off.'

(T13) = (B-INPUT AREA) = 'Any data of B-WORK AREA.'

(T14) = (I-INPUT AREA POINTER) = 'Any number calculated in P 4, 2 of S-PLACE WORD.'

(T16) = (I-POINTER MODIFIER) = '5' or '0.'

APPENDIX E. Extract of "Subprogram Overview Specifications" from DESCRIPTION Program Module of MARSYAS

An overview description of subprograms S-PLACE WORD, S-SCAN CHARACTER, and S-SCAN WORD is given in this Appendix.

S-PLACE WORD

S-PLACE WORD places an argument of MARSYAS statement from B-WORK AREA into an appropriate argument field of B-INPUT AREA for printing program listing.

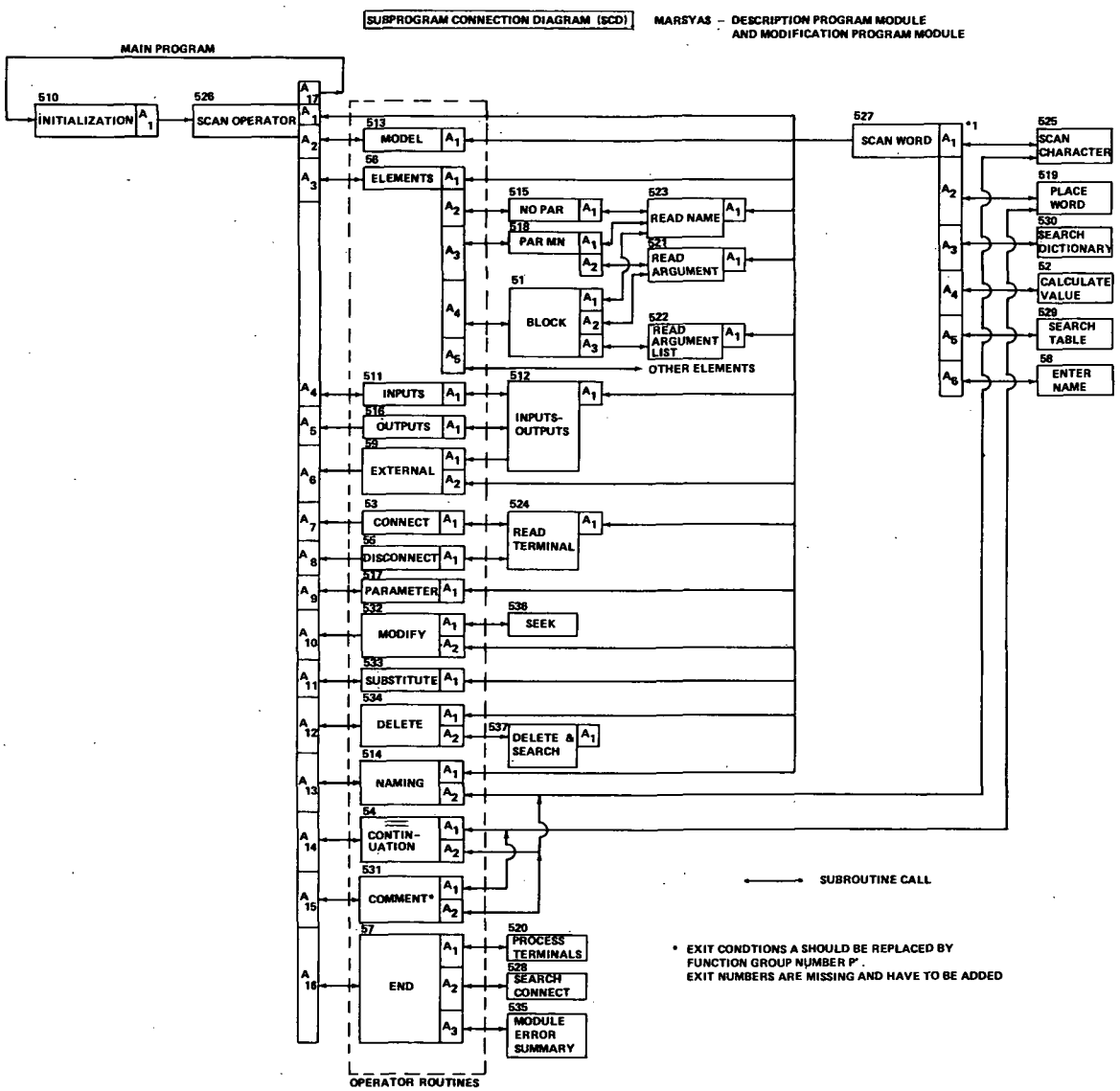
SCAN CHARACTER

SCAN CHARACTER reads the next nonblank alphanumeric character in MARSYAS Language program from the USER DECK into B-WORK AREA.

SCAN WORD

SCAN WORD reads the next word of the MARSYAS-Program from the USER DECK and determines the word type, addresses of operator routines, and status of name. A new name is placed into the T-NAME DICTIONARY together with suffix and subscript information. SCAN WORD places the word into the print area INPUT AREA for the program listing.

George C. Marshall Space Flight Center
National Aeronautics and Space Administration
Marshall Space Flight Center, Alabama 35812, May 23, 1972


SUBPROGRAM CONNECTION DIAGRAM (SCD)
MARSYAS – DESCRIPTION PROGRAM MODULE AND MODIFICATION PROGRAM MODULE
MAIN PROGRAM
510
INITIALIZATION
526
SCAN OPERATOR
513
MODEL
56
ELEMENTS
515
NO PAR
518
PAR MN
523
READ NAME
521
READ ARGUMENT
51
BLOCK
522
READ ARGUMENT LIST
OTHER ELEMENTS
511
INPUTS
516
OUTPUTS
512
INPUTS-OUTPUTS
59
EXTERNAL
53
CONNECT
55
DISCONNECT
524
READ TERMINAL
517
PARAMETER
532
MODIFY
538
SEEK
533
SUBSTITUTE
534
DELETE
537
DELETE & SEARCH
514
NAMING
54
CONTIN-UATION
531
COMMENT*
57
END
520
PROCESS TERMINALS
528
SEARCH CONNECT
535
MODULE ERROR SUMMARY
OPERATOR ROUTINES
527
SCAN WORD
*1
525
SCAN CHARACTER
519
PLACE WORD
530
SEARCH DICTIONARY
52
CALCULATE VALUE
529
SEARCH TABLE
58
ENTER NAME
SUBROUTINE CALL
* EXIT CONDTIONS A SHOULD BE REPLACED BY FUNCTION GROUP NUMBER P'. EXIT NUMBERS ARE MISSING AND HAVE TO BE ADDED

SUBPROGRAM/TABLE AFFECT DIAGRAM (STAD)
(DESCRIPTION MODULE OF MARSYAS)

Subprogram Name	T / S	Tables																		Buffers					Indicators													
		T-Connection Table	T-Elements Table	T-Invalid Connect Table	T-Model Input Table	T-Model Output Table	T-Model Module Table	T-Mnemonic List	T-Name Dictionary	T-Operator List	T-Parameter Table	T-Standard Function List	T-Temporary Connect Table	T-Temporary Disconnect Table	T-Temporary Parameter Table	T-Unconnected Terminal Table	T-Undefined Name List			B-Input Area	B-User Deck	B-Work Area			I-Alpha	I-Attributes Switch	I-Basic ID	I-B Table	I-Character Count	I-Conpointer	I-End-of-Deck	I-Error Indicator 1...N	I-Expected Word Type	I-F Table	I-I	I-ID		
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35			
S-Block	1								0		1														1	1					1							
S-Calculate Value	2																					1							ϕ			1		ϕ			
S-Connect	3												1																			1						
S-Continuation	4																			1												1						
S-Disconnect	5													1																		1	1					
S-Elements	6																															1	1					
S-End	7	ϕ	0	1	0	0			0		ϕ		0	0	0	1	1										1		ϕ			ϕ		1	1			
S-Enter Name	8								ϕ																							1			ϕ			
S-External	9		1				ϕ																				ϕ					1	0					
S-Initialization	10						1																							1								
S-Inputs	11				0																					1					1		1				
S-Inputs/Outputs	12				1	1																				ϕ					1	1	0				
S-Model	13	1			1	1	1		1		1		1	1	1																1	1					
S-Naming	14																														1	1					
S-Nopar	15																								1						1						
S-Outputs	16					0																				1					1		1				
S-Parameter	17										1				0										ϕ						1	1					
S-Parameter	18										1															1	1				1						
S-Place Word	19																			1		0							0								
S-Process Terminals	20		0		0	0																						0		ϕ			0	ϕ		
S-Read Argument	21										ϕ				0												1				1	1					
S-Read Argument List	22										ϕ																				1		0				
S-Read Name	23		1						1																0						1	1					
S-Read Terminal	24																																					
S-Scan Character	25																				0	1									1						
S-Scan Operator	26																			1										0	1	1					
S-Scan Word	27								ϕ													1						1	ϕ			1	0	1				
S-Search Connect	28	0														1																1						
S-Search Table	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0											ϕ					0				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35			

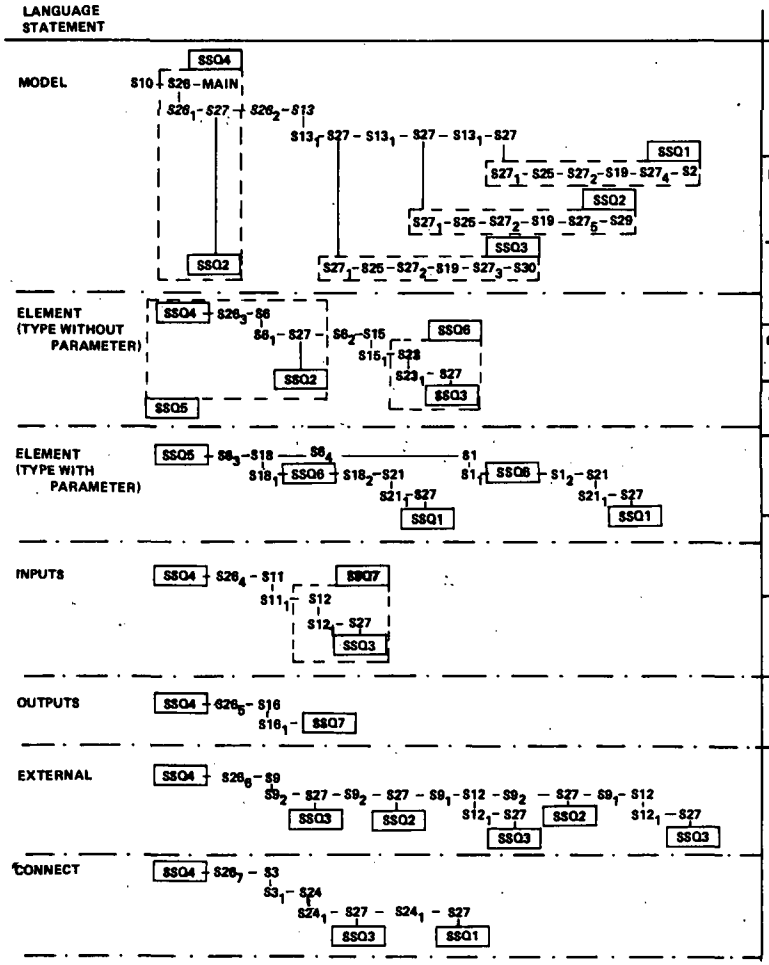
Observations	Comments	Example
1. Column with many 0's and a few 1's	A few subprograms with the 1 in that column affect many other subprograms with the 0.	Column T45: Subprograms S7, 10, and 13 may affect the subprograms S3, 4, 5, 6, 9, 11, 14, 16, and 17.
2. Column with many 1's and a few 0's	Many subprograms with the 1 in that column affect a few other subprograms with the 0.	Column T31: Nearly all subprograms set error indicators (T31) which are processed by only one subprogram S7.
3. Line with many 0's and a few 1's	This subprogram is likely to be affected by many other subprograms.	Line S9: Subprogram S9 may affect only the subprograms S7, 20, and 12; however, it may be affected by subprograms S10, 13, 11,12, 16, 18, 21, 5, 6, 14, 17, 23, 26, 7, 4, 6, 27, 29, 8, and 2.
4. Line with many 1's and a few 0's	This subprogram is likely to affect many other subprograms.	Line S13: Subprogram S13 may affect the subprograms S7, 28, 29, 20, 11, 16, 9, 1, 8, 27, 21, 22, 17, 3, 4, 5, 6, 7, 9, 11, 14, 16, and 17.
5. Line with 1's in columns with no 0's	This subprogram does not affect another subprograms.	Line S19: Subprogram S19 may affect only one other subprogram, S25.

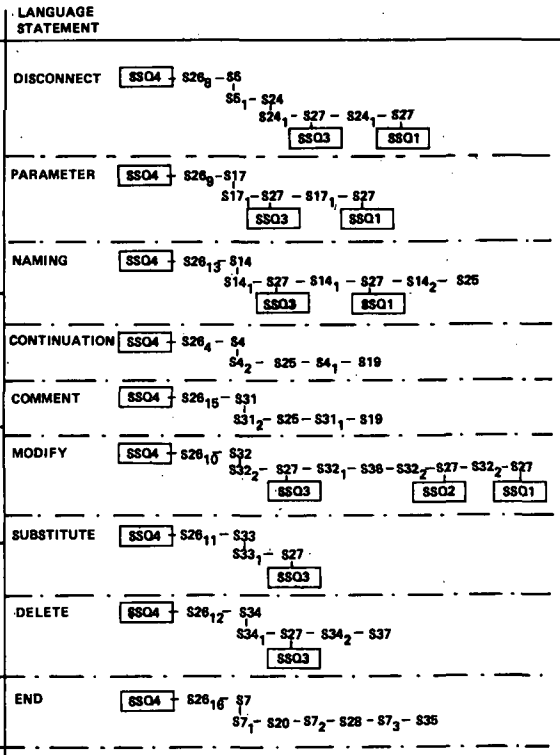
SUBPROGRAM/TABLE AFFECT DIAGRAM (STAD)
(DESCRIPTION MODULE OF MARSYAS)

Subprogram Name	T / S	Indicators																																		
		I-Implied Operator	I-Implied Predecessor	I-Increment	I-Input Area Pointer	I-Length	I-List Switch	I-Match Indicator	I-Model Input (Output) Count	I-Model Sequence Number	I-Module Number	I-NID	I-Number	I-Operator Indicator	I-Paren Count	I-Pointer Modifier	I-Pointer Value	I-Predecessor	I-Result	I-Right Paren	I-SC Pointer	I-Sequence Number	I-Sign	I-Start Disc	I-Statement Number	I-Successor	I-Table Switch	I-TC Pointer	I-TD Pointer	I-Terminator Switch	I-Word Type Switch	I-Work Area Pointer				
		36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
S-Block	1												0																	0	0					
S-Calculate Value	2												ϕ										ϕ								ϕ					
S-Connect	3										0	0																		0						
S-Continuation	4	0									0			1																						
S-Disconnect	5										0	ϕ																		0						
S-Elements	6										0			1																0						
S-End	7										ϕ													1				ϕ	1	0						
S-Enter Name	8																														1					
S-External	9						1				0			0	0												1			0	0					
S-Initialization	10										1										1	1			1											
S-Inputs	11						1				0																1									
S-Inputs/Outputs	12						0																				0			0					
S-Model	13										ϕ																			0						
S-Naming	14										0	ϕ									ϕ									0	0					
S-Nopar	15																													0						
S-Outputs	16						1				0																1									
S-Parameter	17										0	1	0																	0	0					
S-Parameter	18																													0						
S-Place Word	19				ϕ									0		1				0												1				
S-Process Terminals	20		ϕ						0			ϕ					1								1					0					
S-Read Argument	21																																			
S-Read Argument List	22														0																					
S-Read Name	23																					1														
S-Read Terminal	24																																			
S-Scan Character	25																				ϕ											0			
S-Scan Operator	26				1										1										1										
S-Scan Word	27	1		1		1								1	ϕ					1	1									1	ϕ	1				
S-Search Connect	28																																			
S-Search Table	29			0				1						1																	1					
		36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		

MARSYAS — DESCRIPTION PROGRAM MODULE AND MODIFICATION PROGRAM MODULE

SUBPROGRAM SEQUENCE DIAGRAM (SSD)





SUBPROGRAM NUMBER LIST

NO. S	NAME
S 1	S-BLOCK
2	S-CALCULATE VALUE
3	S-CONNECT
4	S-CONTINUATION
5	S-DISCONNECT
6	S-ELEMENTS
7	S-END
8	S-ENTER NAME
9	S-EXTERNAL
10	S-INITIALIZATION
11	S-INPUTS
12	S-INPUTS – OUTPUTS
13	S-MODEL
14	S-NAMING
15	S-NOPAR
16	S-OUTPUTS
17	S-PARAMETER
18	S-PARMN
19	S-PLACE WORD
20	S-PROCESS TERMINALS
21	S-READ ARGUMENT
22	S-READ ARGUMENT LIST
23	S-READ NAME
24	S-READ TERMINAL
25	S-SCAN CHARACTER
26	S-SCAN OPERATOR
27	S-SCAN WORD
28	S-SEARCH CONNECT
29	S-SEARCH TABLE
30	S-SEARCH DICTIONARY
31	S-COMMENT
32	S-MODIFY
33	S-SUBSTITUTE
34	S-DELETE
35	S-MODULE ERROR SUMMARY
36	S-SEEK
37	S-DELETE & SEARCH

NOTE: BARS ABOVE SUBPROGRAM NUMBERS DESIGNATING LOOPS ARE OMITTED.

REFERENCES

1. Wanted for the 70's: Easier-to-Program Computers. Special Report, Electronics, September 13, 1971, p. 62.

2. Boehm, B. W.: Some Information Processing Implication of Air Force Space Missions in the 1970's. Astronautics & Aeronautics, January 1971.

3. UNIVAC Systems Programming: Vocabulary for Information Processing. Sperry Rand Corp., 1968.

4. MARSYAS-User Manual. Interim Version, MSFC, Computation Laboratory, July 1, 1971.

5. MARSYAS Language Specifications. Up-dated Version, Computer Applications, Inc., March 1970.

6. Prasad, N. S., and Reiss, J.: The Digital Simulation of Interconnected Systems (Up-dated Version). Computer Applications, Inc., September 1970.

7. Prasad, N. J., and Gabow, H.: ADEPT-An Algebraic and Differential Equations Processor and Translator. Systems Consultant, Inc., June 1971.

8. Trauboth, H., and Prasad, N.: MARSYAS-A Software System for the Digital Simulation of Physical Systems. Proc. of Spring Joint Computer Conference, May 1970.

9. Trauboth, H., and Prasad, N.: MARSYAS-A Software Engineering System for the Digital Simulation and Analysis of Physical Systems. Proc. of IFAC Symposium on Digital Simulation of Continuous Process, Budapest, Hungary. September 1971.

10. Marshall Information Retrieval and Display System (MIRADS), A Data Management System, User's Mannual. Computation Laboratory, Computer Science Corporation, Doc.No. MA-010-001-2H.

11. APOLLO Configuration Management Manual. Exhibit XVIII, NHB 8040.2 (formerly NPC 500-1), NASA, Office of Manned Space Flight, January 1970.

☆ U.S. GOVERNMENT PRINTING OFFICE: 1973 - 736-410, REGION NO. 3-11

"The aeronautical and space activities of the United States shall be conducted so as to contribute . . . to the expansion of human knowledge of phenomena in the atmosphere and space. The Administration shall provide for the widest practicable and appropriate dissemination of information concerning its activities and the results thereof."

—NATIONAL AERONAUTICS AND SPACE ACT OF 1958

NASA SCIENTIFIC AND TECHNICAL PUBLICATIONS

TECHNICAL REPORTS: Scientific and technical information considered important, complete, and a lasting contribution to existing knowledge.

TECHNICAL NOTES: Information less broad in scope but nevertheless of importance as a contribution to existing knowledge.

TECHNICAL MEMORANDUMS: Information receiving limited distribution because of preliminary data, security classification, or other reasons. Also includes conference proceedings with either limited or unlimited distribution.

CONTRACTOR REPORTS: Scientific and technical information generated under a NASA contract or grant and considered an important contribution to existing knowledge.

TECHNICAL TRANSLATIONS: Information published in a foreign language considered to merit NASA distribution in English.

SPECIAL PUBLICATIONS: Information derived from or of value to NASA activities. Publications include final reports of major projects, monographs, data compilations, handbooks, sourcebooks, and special bibliographies.

TECHNOLOGY UTILIZATION PUBLICATIONS: Information on technology used by NASA that may be of particular interest in commercial and other non-aerospace applications. Publications include Tech Briefs, Technology Utilization Reports and Technology Surveys.

Details on the availability of these publications may be obtained from:

SCIENTIFIC AND TECHNICAL INFORMATION OFFICE

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

Washington, D.C. 20546